Using 3D functionality available in current web-browsers to create and visualize geological models.

Øystein Ivar Malt

Master's thesis in Software Engineering at

Department of Computing, Mathematics and Physics,
Western Norway University of Applied Sciences

Department of Informatics,
University of Bergen

June 2017

Abstract

This thesis investigates the possibility of using modern web technologies to develop accessible applications for interactive covisualization of geological data such as topography, seismic slices and measurements from wells. To avoid low-level development we specifically investigate X3DOM, which hides the details of graphics rendering in a high-level, declarative XML-like syntax. The thesis shows how to successfully implement a geoscientific application using X3DOM and the Angular web application framework.

Acknowledgements

I would like to thank my advisor at CMR, Daniel Patel, and my supervisors from Bergen University College, Harald Soleim and Atle Geitung. Without their invaluable input, continued support and guidance this thesis would not have been possible.

Table of contents

Table of figures

Table of listings

1 Introduction

1.1 Problem description

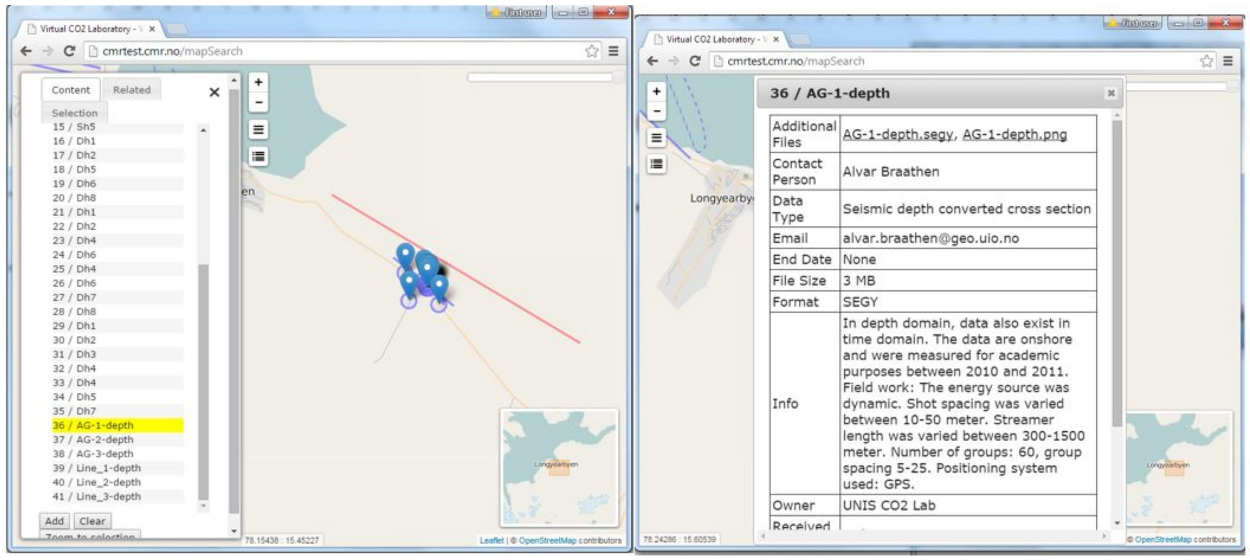
We want to investigate the possibility of visualizing subsurface geological data using modern web technologies, specifically the X3DOM framework.

There are a number of applications for visualizing 3D geological subsurface data in existence today. These applications are usually proprietary, requiring costly licenses, in addition to having a steep learning curve [5]. This means that users have to invest in expensive application suites and extensive training, which may result in vendor lock-in. Another issue with current solutions, is that they require installation on the specific computer in use. This can affect collaboration with other parties, as all collaborators have to install the application before collaboration can commence. Lastly, the complexity of developing the applications themselves is an issue. Existing applications are developed, relying on low level graphics libraries such as OpenGL and DirectX. This can cause applications to require much time to develop and maintain, which in turn may contribute to their expensive pricing.

We wish to investigate the possibility of using modern web technologies to develop accessible applications for visualizing large scale geological data. To avoid low-level development we specifically investigate X3DOM, which hides the details of graphics rendering in a high level, declarative XML-like syntax.

Using the 3D capabilities built into modern web browsers with WebGL, 3D applications that will run in browsers on any platform and device can be created, eliminating the need for developers to address cross platform support. These types of applications can help facilitate collaboration between different parties as there is no need to install anything to access the application, one can simply navigate to a web page to do so.

The motivation for this thesis comes from Christian Michelsen Research's (CMR) Virtual CO_2 Laboratory project (VIRCOLA) [5]. The project is tied to the National Center for Environmental-friendly Energy, SUCCESS (Subsurface CO2 Storage- Critical Elements and Superior Strategy). The goal of the VIRCOLA project is to "develop a data platform and methodology that can facilitate better data utilization and work processes, and lead to better understanding of the storage capacity, injectivity and long term confinement of CO2" [5]. As a part of this project, CMR used the industrial grade geo-scientific 3D visualization application SKUA. It was experienced that SKUA was more suited for detailed interpretation as opposed to getting an overview of existing data and having a fast and simple navigation. Because of this, and with SKUA being software with a high price and high learning curve, CMR created an easy-to-use, free 2D downscaled web application that was accessible for anyone with a web browser [6], see *Figure 1*. This web application allowed users to get a quick overview of the positioning of available geological and geospatial data, along with related research work, leaving detailed visual analysis to professional applications such as SKUA and Petrel (Another industrial geo-visualization application, see section 1.2). In the web application (see *Figure 1* left), data was annotated on a 2D map as blue points (e.g. wells), or red lines (e.g. seismic slices). By clicking on an annotation, extra information was shown and the corresponding data could be downloaded (*Figure 1* right), such as seismic data, well data and accompanying publication reports [6].



36 / AG-1-depth

Additional Files	AG-1-depth.segy, AG-1-depth.png
Contact Person	Alvar Braathen
Data Type	Seismic depth converted cross section
Email	alvar.braathen@geo.uio.no
End Date	None
File Size	3 MB
Format	SEGY
Info	In depth domain, data also exist in time domain. The data are onshore and were measured for academic purposes between 2010 and 2011. Field work: The energy source was dynamic. Shot spacing was varied between 10-50 meter. Streamer length was varied between 300-1500 meter. Number of groups: 60, group spacing 5-25. Positioning system used: GPS.
Owner	UNIS CO2 Lab
Received	

Figure 1: The CMR 2D application. To the left, a map view can be seen, with data shown as blue points or red lines. To the right, an information window for a data item can be seen.

All the 3D data in the application was downloadable from the 2D topographical map but was not visualized in the application itself as it only supported 2D. Because of this, a web application that is able to visualize seismic slices and well data directly in a topographical 3D terrain was desirable, which in addition was built for achieving quick overviews instead of detailed problem solving and analysis like SKUA and PETREL. The requirements of such an application, provided by CMR, was that the relevant data should be visualized at correct geospatial coordinates, including depth. The application should contain a topography of the area where the data resides. It was also important that the visualizations could easily be toggled on and off, as they often overlap each other. CMR was not able to do this easily in SKUA, which caused them to spend time during interaction when trying to hide or display specific data items. Publications and reports that are relevant to a single visualized data item should also be accessible by interacting with the visualization, which also was difficult to do in SKUA [5]. An application with the functionality described above, that is simple to use and accessible in a web browser does not seem to be available in the market. A similar observation was done in 2011 by Nimtz et al. [41]. Thus such software would likely be beneficial for the geoscientific community.

This thesis also investigates the X3DOM framework's viability for use in applications visualizing 3D subsurface geological data in a large area. The framework has the strength of being built around the X3D ISO-standard, which is expected to be supported for a long time. It is also declarative, meaning that developers can describe a 3D scene on a high level, and are not required to learn the low level WebGL API or the GLSL shader language in order to create applications [7]. This thesis will provide insight into the attempt at creating a geologic visualization application using X3DOM, and may be useful for developers evaluating X3DOM in other projects.

1.2 Related work

This thesis aims to investigate visualization of subsurface geological data using modern web technologies, specifically X3DOM. There has been related work using X3DOM for geospatial visualizations in web browsers. JavaScript web frameworks such as Cesium has been used for subsurface visualizations. Also there exist feature rich applications for desktops made specifically for subsurface visualizations, among them SKUA and Petrel.

Ziolkowska and Reyes' paper [8], "Geological and hydrological visualization models for Digital Earth representation" from 2016 presents interactive visualization models for geo-temporal and geospatial data in virtual globes. The authors use KML (Keyhole Markup Language) to visualize data on a virtual globe. There are several virtual globes available that supports KML, notably Google Earth, NASA WorldWind and Cesium. The article comments on subsurface visualization in virtual globes, noting that these "were not originally developed with the purpose of representing data below the Earth's surface" [8]. From experience with using the different virtual globes, navigating below the surface of the globe is hard, as support for this is not made. Adding geometry below the surface can be done but this is subsequently overdrawn, and solving this problem would require access to low-level rendering details which are not clearly exposed.
The models presented in the paper [8] circumvents these limitations by introducing transparent earth layers around the globe, created using KML, treating the top layer as the surface of the earth, and thus enabling the camera to move "subsurface" to view the geospatial visualizations. Due to this "hack", the real globe rendered by the respective framework can be seen as an artifact at an arbitrary depth inside the transparent layers.

Krämer and Gutbell's paper [7] "A case study on 3D geospatial applications in the Web using state-of-the-art WebGL frameworks" from 2015 investigates the open-source frameworks

Cesium, Three.js and X3DOM for use in geospatial applications in web browsers. The authors give a qualitative comparison of the frameworks based on several software prototypes developed to assess them. They find that each framework has different approaches, goals and target groups. Cesium is targeted specifically at geospatial applications, and is well suited for such. Three.js offers direct access to WebGL, making it suitable for a wide range of use-cases due to this flexibility, although developers might have to implement desired features. X3DOM is declarative, and based on the standardized X3D file format. This has the advantage of developers not having to learn how to use the low-level WebGL API, and since it is built on a standard it is also suitable for applications required to be supported for a long time.

CMR has evaluated visualization applications Petrel and SKUA for use in the visualization of geological data tied to the SUCCESS research project. In their article, “VIRCOLA - Review of Data and Visualization Platform” [5], they discuss the strengths and weaknesses of each application for this use-case. The report finds that both these applications can be used to visualize all data from the SUCCESS project. Petrel is a software platform containing visualization software for a wide range of geological and petroleum related data. It is used for analysis and decision making both during exploration and production of oil wells. Using Petrel geology can be modeled and analysed, along with well correlation and seismic data, to name a few [37]. SKUA is a visualization and modeling software suite developed by Paradigm. It contains features such as seismic processing and imaging, interpretation of seismic and geological data, modeling of subsurface and reservoir data [38]. However, both these systems are costly, require installation and do not run in a browser.

2 Background

2.1 WebGL

WebGL is a 3D graphics API available on the web platform. It enables the browser to use the underlying computer's hardware to render graphics. WebGL is a low-level API that has to be supplied with arrays of data and shaders for rendering the data. This is done using WebGL's JavaScript interfaces to pass data to the GPU (Graphics Processing Unit) and perform render calls. There are, however, a large amount of frameworks and libraries that have been built on top of WebGL to provide a more high-level interface to the API. WebGL was created at Mozilla in 2006, with the intention of providing a 3D graphics API for the Canvas HTML element. The design was based on OpenGL ES 2.0, which is a subset of OpenGL meant for embedded systems. The WebGL specification has since 2009 been maintained by the Khronos Group. It is not part of the official HTML5 specification, but is now supported by all major browsers, and has become the defacto standard API for 3D graphics in browsers. [4]

To use WebGL in a web application, an HTML5 canvas element must be created by declaring a *<canvas>* element in the HTML code, which defines the drawing area for graphics. The drawing context for WebGL must then be retrieved from the canvas using JavaScript, and the bounds of the drawing area (called a viewport) must be set. WebGL uses primitives to draw geometry. Primitives are objects that can be either triangles, triangle strips, points, or lines. These primitives are stored in arrays called buffers, where e.g. the positions of the edges (vertices) of triangles are defined. Before the data in the buffer can be used to draw geometry, two matrices must be created. A model-view matrix that translates, rotates and scales vertices from model coordinate space (where they initially were declared) to the 3D coordinate system relative to the camera (view space), and a projection matrix, that transforms the 3D coordinates of the model into the 2D coordinates of the viewport.
The last thing required to draw with WebGL is shaders. Shaders are small programs written in the OpengGL Shading language (GLSL), which is a low-level C-like language. The shaders define how the pixels for 3D objects actually get drawn on the screen. At least two shaders are required for making graphics, a vertex shader and a fragment shader. The vertex shader gets the vertices defined in the buffer as input, and is required to transform them to 2D screen space. The fragment shader is used to generate the color of each individual rasterized pixel from the vertices transformed in the vertex shader. Buffers containing data for colors on a per-vertex basis, textures and more can be used as input to the vertex shader to generate the desired color for each pixel.

The shaders must be compiled to the GPUs instruction set and the shaders must then be initialized with required input (such as uniforms and buffers), before at last the graphics can be drawn [4].

2.2 Choosing a graphics framework/library

There exists a large amount of libraries and frameworks with the purpose of making web based graphics easier to develop. As there are such a large amount of options, this discussion will concentrate on X3DOM and Three.js, since the former is the focus of this thesis and the latter is one of the most popular JavaScript libraries for WebGL.

Cesium, Google Earth and NASA WorldWind are other important frameworks in this context, due to their extensive support for geospatial data. However, these frameworks do not readily support subsurface visualizations, as discussed in section 1.2. It is possible to circumvent this as shown in [8]. However, the concern of using this approach for the thesis is that the additional transparent spheres could potentially interfere with the detailed visualizations. These frameworks were therefore not evaluated further.

2.2.1 Three.js

Three.js is an open-source JavaScript library for creating 3D graphics, built on top of the low level WebGL API. Three.js makes creating graphics in the web browser easier. It contains abstractions for common tasks in WebGL and provides, among others, access to features such as [2]:

- Managing objects in a scenegraph.
- Rendering the scenegraph using WebGL, SVG or even CSS (with the purpose of supporting browsers without WebGL).
- 3D related math, e.g. bounding boxes, transformations and rotations on scene objects.
- A very large amount of predefined objects in the framework, such as Geometries, Materials (Shaders), Lights, Camera, Graphical effects, and Loaders for 3D models.

A simple example of a Three.js application for displaying a shaded sphere can be seen below. *Code Listing 1* shows the JavaScript file containing the Three.js code, while *Code Listing 2* shows a minimal HTML5 file for the application. When the web page's load event fires, the class constructor is run, setting up the application. Notice that there are several objects that need to be initialized before the sphere can be created. Three.js provides easy to use Camera, Scene and Renderer objects, abstracting quite a lot from WebGL. These three objects are always used in a Three.js application (PerspectiveCamera is a subclass of Camera, and WebGLRenderer is the subclass of Renderer that uses WebGL). After these have been initialized, two lights are created using Three.js' AmbientLight and PointLight types, and added to the scene. The sphere is then created by using the library's SphereGeometry type, and combining this with a MeshPhongMaterial (which adds support for light on the shape) to create a Mesh object. This Mesh is added to the scenegraph, and the rendering loop is started.

```
class App {
    constructor() {

        // Configure Three.js
        this.width = 800;
        this.height = 640;
        this.camera = new THREE.PerspectiveCamera(45, this.width / this.height, 0.1, 1000);
        this.scene = new THREE.Scene();
        this.renderer = new THREE.WebGLRenderer({
            antialias: true
        });
        this.scene.add(this.camera);
        this.camera.position.z = 50;
        this.renderer.setClearColor(0xFFFFFF);
        this.renderer.setSize(this.width, this.height);
        document.body.appendChild(this.renderer.domElement);

        // Adding light
        let ambientLight = new THREE.AmbientLight(0xFFFFFF);
        let pointLight = new THREE.PointLight(0xFFFFFF, 2, 100);
        pointLight.position.set(0, 0, 40);
        this.scene.add(pointLight);
        this.scene.add(ambientLight);

        // Create the sphere, and add it to the scene
        let sphereGeometry = new THREE.SphereGeometry(5, 16, 16);
        let sphereMaterial = new THREE.MeshPhongMaterial({ color: 0x0000FF });
        let sphere = new THREE.Mesh(sphereGeometry, sphereMaterial);
        this.scene.add(sphere);

        this.render();
    }

    render() {
        this.renderer.render(this.scene, this.camera);

        // Bind the context so the content of the class can be accessed!
        window.requestAnimationFrame(this.render.bind(this));
    }
}

window.onload = new App();
```

Code Listing 1: A basic Three.js example.

```
<!DOCTYPE html>
<html>
    <head>
        <script src="/path/to/three.js"></script>
        <script src="/path/to/my-application-file.js"></script>
    </head>
    <body></body>
</html>
```
Code Listing 2: A minimal HTML5 file loading the Three.js library and the application from Code Listing 1.

The result of *Code Listing 1* and *Code Listing 2* can be seen below in *Figure 2*.

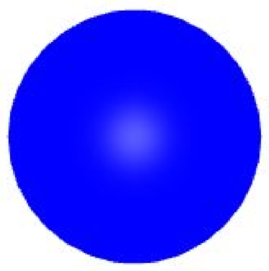


Figure 2: The result of Code Listing 1 and 2. A blue shaded sphere rendered with Three.js.

2.2.2 X3DOM and X3D

X3D is an XML-based file format for representing 3D computer graphics in a declarative manner, and is an ISO standard. It describes a file format, a scene- and an event-graph. X3D scenes are encoded using XML syntax, derived from its predecessor VRML97 [1].

X3D is only a specification. X3DOM is an implementation of this specification, along with support for regular DOM interaction with the X3D elements declared in the HTML file. The X3DOM framework has support for most major browsers, both desktop and mobile versions [39]. This means that most platforms and operating systems are supported inherently by the framework.

X3DOM supports X3D data in HTML5 content, inside the browser's Document Object Model (DOM) tree. The DOM is a model with an API that represents the HTML text document as a tree structure where each node represents an element in the document. The DOM allows programs and scripts to "dynamically access and update the content, structure and style of documents" [14]. *Figure 3* can help get an understanding of what X3DOM is by showing it in relation to 2D HTML canvas drawing, SVG (Scalable Vector Graphics) and WebGL. For drawing 2D graphics, imperative commands can be executed with the 2D canvas. Alternatively the graphics can be defined declaratively using the SVG vector format. For 3D graphics, the parallel to the canvas is to use WebGL and issue imperative draw calls. However if one wants to define the 3D graphics declaratively, possibly from a file format, one can then use X3DOM.

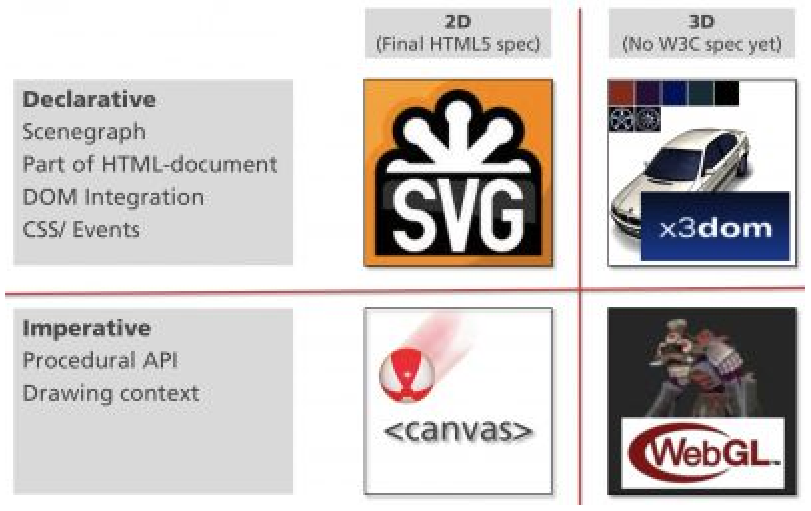


Figure 3: X3DOM compared to 2D imperative vector drawing, SVG and WebGL (from [16]).

X3DOM "acts as a connector for the HTML5 and X3D world and content." [1]. The connector is at the core of X3DOM and is responsible for binding the X3DOM tree, defined in the browser's DOM, to the X3D backends and support functionality. X3DOM does not directly render the X3D elements in the DOM-tree, but maintains a representation of an X3D-tree that is synchronized with the X3DOM nodes from the DOM-tree [1]. X3DOM is open source and dual-licensed under the MIT and GPL license [15].

Because of the integration with the DOM, the same operations that are possible to execute on regular HTML elements are also possible on the X3DOM elements. The DOM updates are reflected in the X3D representation of the scene by the framework. An X3DOM element can be accessed by using the DOM's *getElementById* function. The element can then have its attributes changed by using regular DOM functionality. Attributes specific to X3D elements (e.g. diffuseColor, translation, orientation, etc.) can also be changed in this manner.

The X3DOM runtime works by scanning the HTML document for an *<x3d>* element tag. When this is found, it reads the content of the tag, using the DOM to access it. An HTML canvas element is added to the DOM, reading attributes from the *<x3d>* tag to specify, among others, the width and height of the canvas' drawing surface. The contents of the *<scene>* element tag (inside the *<x3d>* tag) are then read and used to build the scenegraph to be displayed on the canvas [42].

A simple example of an X3DOM scene can be seen in *Code Listing 3* below. The *<x3d>* element describes the X3DOM context that will contain the graphics. A *<scene>* element declares a scenegraph, containing all the elements declared inside it. The *<transform>* element is used to create a coordinate system for the children of the element, relative to the coordinate system(s) of its ancestor(s). In this example the transform element is used to control the position of the *<shape>* element, although it does not change the translation from the center of the scene here (it is only added for informational purposes). The *<shape>* element defines a 3D object. This element contains the description of how the shape looks. The geometry of the shape is specified, stating that it is a sphere. The appearance of the shape is also specified, adding a material with a blue color (the diffuseColor attribute accepts RGB input, with each channel containing a decimal value between 0-1).

```
<html>
...
<x3d width="800px" height="640px">
    <scene>
        <transform translation="0 0 0">
            <shape>
                <appearance>
                    <material diffuseColor="0 0 1"></material>
                </appearance>
                <sphere></sphere>
            </shape>
        <transform>
    </scene>
</x3d>
...
</html>
```

Code Listing 3: A basic X3DOM example, creating a blue sphere.

The result of *Code Listing 3* can be seen below in *Figure 4* where a blue sphere is rendered.

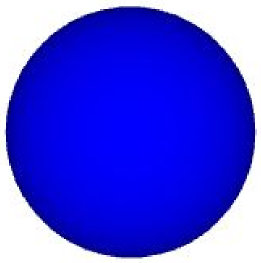


Figure 4: The result of Code Listing 2, rendering of a blue sphere with default X3DOM shading.

2.3 Web application frameworks

To create the web application in this thesis, we wanted to use a framework to maximize code reuse and simplify development of application logic. JavaScript has become an increasing part of web development, and modern web applications have become more complex due to increase of users and demands. To make development faster and simpler, developers have created libraries and frameworks with utilities and functions to more easily handle creation and management of user-interfaces, event handling, application logic, etc. [9].

There are a large amount of libraries and frameworks for general web development in existence. Based on the number of tags on posts at Stack Overflow [47], two of the most used web frameworks today are Google's Angular (page 3 at [47]) and Facebook's React (page 4 at [47]). Because there is an abundance of web frameworks, only these two are considered in this thesis to limit the scope.

2.3.1 Angular

Angular is the successor of Google's AngularJS framework, which rose to high levels of popularity in recent years. Angular is a framework for building dynamic web applications, it focuses on modularity, and uses components as the core building block of user interfaces. Angular uses TypeScript, which is a superset of JavaScript that provides features from future JavaScript standards and optional static type checking [10].

An Angular component defines and controls a part of the user interface, called a *view*. A component's application logic resides in a TypeScript class. The interaction between the logic and the view is connected through an API of properties and methods in this class. The appearance of the view is declared in the component's associated *template*. The template defines the HTML code for the component, and is written in a specialized form of HTML specific to Angular. The template can either be declared in a separate file, or written as a string directly in the component's decorator [48]. A decorator is used to provide Angular with metadata, describing how the class should be processed. The component class is a standard TypeScript class, but contains the "@Component" decorator from the Angular framework. Angular applications are developed by creating components that display and control small UI elements. These are combined to create larger views, making up a section of the complete UI. Lastly, these larger views are combined inside a root component for the entire UI that serves as the entry point of the Angular application [10]. *Figure 5* below visualizes this composition of components.

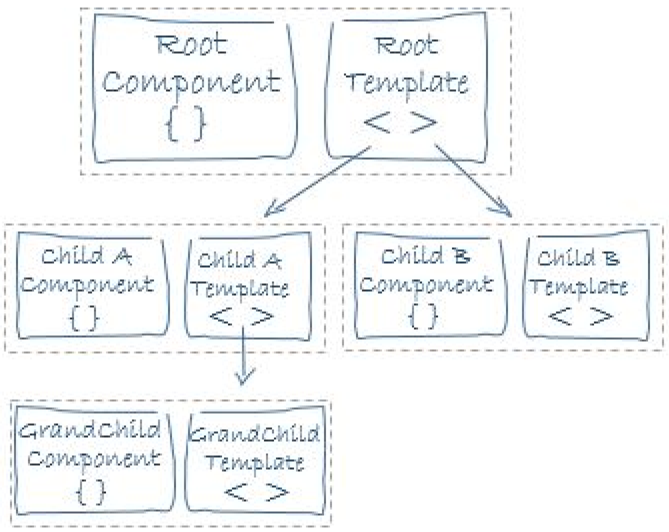


Figure 5: Component composition of an Angular application (from Angular's documentation [10]).

In addition to components, Angular applications can take advantage of structures called services and directives. The Angular service is a broad category meant to contain code to provide the application with functions or data that are not directly tied to the view logic. An example of code belonging in a service is code that fetches data from a server. These services can be injected into components as needed by using Angular's dependency injection system.
The Angular directive is in essence a component without a view. In fact, a component is technically a directive. When Angular renders templates for views, it does so as per instructed by directives. Directives can be added to template elements to modify their appearance or functionality [10]. Angular is a large framework with complex functionality which can not be adequately described here. For more information see the Angular documentation at [40].

2.3.2 React

The second web framework we evaluated in this thesis is React. React is a JavaScript library for simplifying the process of creating user interfaces. Like Angular, React calls the building block of the UI for components. These components are also combined in a similar manner as in Angular [11].

The largest difference between React and Angular is that React is a library, while Angular is a framework. React only concerns itself with rendering and controlling the views, while Angular has additional functionality for building complete applications, such as routing and http. The approach to UI creation is also different. Where Angular components' views are created using string-based templates (see section 2.3.1), meaning you write the template inside a string, React components' views are created using JavaScript only, with the option of using JSX (a language extension for JavaScript) for declaring the view using HTML syntax directly inside the JavaScript code. React recommends using JSX for creating components. JSX allows the developer to write HTML tags mixed with JavaScript expressions directly in the JavaScript code (e.g. "return <p>Hello World!</p>"). When compiled, the JSX expressions are transformed into JavaScript objects [13], which React uses to create the view.

The use of components lets developers organize the user interface of an application in independent, reusable pieces. A component in React is just a JavaScript class or function that returns React elements describing what to display. A React component receives as input a *props* object, and returns a component. The *props* object holds data defined programmatically, or if JSX syntax is used to instantiate the element, the *props* object holds the properties and attributes declared on the element. React components can be defined in both functions and JavaScript 6 classes, as long as it accepts the *props* object, and returns a component. In the case of a class component, the *props* object is passed as an argument to the constructor of the class, and is stored as an instance variable in the class. The *render* function in a React class component returns the view of the component, and is required by the framework [11]. Similar to Angular, React is a large library which is hard to sufficiently explain in the scope of this thesis. For more information see [11].

In the example in *Code Listing 4* below, a simple React component is created with JSX that takes a JavaScript array of names and translates them to a displayable HTML list of names. A collection of list items are built from the *names* property on the *props* object of the component by using the *map* function of JavaScript. The JavaScript *map* function executes the callback function provided as an argument for each element in an array, with the current element passed as an argument to the function, and creates a new array containing the values returned from the function [12].

The curly brackets in JSX syntax can contain any valid JavaScript expression, and the result of the expression is put into the HTML. In the example, the *map* function is used to create the HTML <*li*> elements, and JSX syntax is used in the callback function to create the individual elements, populating them with names. The return statement in the *render* function returns the <*ul*> element that wraps the <*li*> elements, and the variable containing these elements are evaluated using the curly bracket syntax of JSX [13].

```
export class NameList extends React.Component {
  render() {
    const names = this.props.names.map( name => {
       return (
          <li>{name.last}, {name.first}</li>
       );
    });

    return (
       <ul>{names}</ul>
    );
  }
}
```

Code Listing 4: A simple React component with JSX, where a list of names is rendered.

The NameList component can be used in other components as seen below in *Code Listing 5*. Note here that the *names* property is used to pass an array of objects containing names to the component. This value is passed to the NameList component's constructor, as part of the *props* object by React.

```
class App extends React.Component {
  render() {
    const names = [{first: 'John', last: 'Johnson'}, {first: 'Jane', last: 'Doe'}];

    return (
       <NameList names={names} />
    );
  }
}

// render() will return the html string:
// <ul>
//   <li>Johnson, John</li>
//   <li>Doe, Jane</li>
// </ul>
```

Code Listing 5: Displaying how the NameList component from Code Listing 4 can be used.

3 Problem analysis and decisions made

In the previous chapter, the graphical framework X3DOM and the Three.js library were introduced. The web application framework Angular, and the library React were also presented. In this chapter we make decisions on which of these to choose for development of the application.

3.1 X3DOM vs Three.js

As shown in the previous chapter, X3DOM and Three.js represent two different approaches to browser based 3D graphics. Three.js uses the traditional imperative programming model relying on JavaScript as a programming language, and utilizing WebGL directly through wrapped API calls. X3DOM on the other hand, is a declarative framework. This means that instead of programming and calling functions to render graphical elements to the screen, these elements can be declared using XML-like syntax directly in the HTML code. While Three.js is simple to use and develop applications with, there is still more configuration and code involved with writing an application using this technology. X3DOM might be thought of as an order of abstraction above Three.js, considering no rendering details are handled directly by the developer (when disregarding the option to programmatically create custom X3DOM nodes which requires low level and imperative programming). As can be seen in the examples in *Code Listing 1 & 2* vs *Code Listing 3*, the Three.js code to achieve the same results as with X3DOM is more than four times as long (41 lines versus 8 lines). The inherent simplicity and readability of X3DOM code when compared to Three.js, and the fact that Three.js requires direct involvement with a greater number of variables connected to the graphical elements, can be said to support the statement that X3DOM is easier to use and has a lower risk of logical errors during development. Additionally, X3DOM is based on an ISO-standard, which lends credibility to it as a framework and makes it well suited for industrial applications. Based on these arguments, X3DOM was chosen to be investigated in this thesis due to its potential for robust and fast web graphics development.

3.2 Web application frameworks and X3DOM

The standard method for developing X3DOM applications, is adding the X3D nodes describing the graphics directly in an HTML page [16]. If the properties of the nodes needs to be changed during runtime, the DOM functionality of the browser is used to access individual X3D nodes, and update property values. When the number of nodes in the application becomes large, the X3D code can be stored in separate X3D files, which are included in the X3DOM scene using the *inline* node. However the content of *inline* nodes is not reachable using DOM functions. This has the unfortunate effect of disallowing runtime changes of the loaded X3D models.

Since inline nodes are not dynamically changeable, a solution would be to write the entire application in a single HTML file. However, for a larger application where X3DOM is used to display visualizations of data fetched from a server, declaring the entire graphical application directly in the HTML file is not practical. It would also mean creating a single monolithic file, which could be challenging to maintain and debug. An alternative approach, which was chosen in this thesis, is splitting up the X3DOM code and inserting each part at runtime, as the DOM supports runtime insertion of elements. This allows dynamic generation of X3DOM code, where data fetched from a server can be used to generate code, which can then be inserted into the DOM.

To make X3DOM more efficient to use for development of a visualization application containing a large amount of repetitive geometry, it was decided to attempt to use a web framework to encapsulate X3DOM logic. This would enable the use of these framework's features to create a component based architecture for the application, and also enabling reuse of X3DOM logic. We were not able to find published attempts at this, so if this could be done it could have value for future applications developed using X3DOM.

As mentioned earlier, X3DOM works by inserting X3D elements into an HTML page, which the framework then uses to render graphics. When being used with application frameworks like Angular or React, which also introduce non-standard HTML element tags into the page (called components by these frameworks), issues appear because these frameworks restrict usage of custom elements that do not belong to the respective framework. This leads to X3DOM element tags not being added to the DOM properly, and therefore X3DOM is not able to properly render them.

When using web frameworks that use data and event bindings, the X3DOM framework does not trigger event listeners that are added to encapsulated components in such a manner. Since the X3DOM elements are rendered inside an HTML canvas, events connected to the canvas element are processed by X3DOM. If an event listener is added to an X3DOM node in the scene, X3DOM tracks the related event (e.g. a click event) in the scene and notifies the event listener. To support standard DOM event listener functionality, X3DOM overrides the DOM's default *addEventListener* function on all elements declared inside the <*x3d*> element tag [43]. As this happens first when X3DOM has been loaded, from experience with the web frameworks evaluated in this thesis, it does not appear to be possible to bind to X3DOM events using the web frameworks' conventional approaches. To circumvent this issue, event listeners must be added to X3DOM components using DOM functionality after the web framework has added its components to the DOM. Angular and React both provide component lifecycle events, for instance an event when the component has been added to the DOM. This lifecycle event can be used for setting up X3DOM event listeners properly.

3.2.1 React

React allows any element tag to be added to its views, however it only allows whitelisted properties and attributes on the elements [50]. This presents a problem as X3DOM specific attributes are not in this whitelist. React removes properties and attributes that are not in the whitelist before adding views to the HTML page, thereby not allowing necessary configuration of X3DOM elements. JSX has support for a keyword on its elements, *is*, which bypasses the whitelist. This appears to be the only way to get X3DOM working in React without adding functionality to circumvent the whitelist. The negative side of using this approach is that it is not documented. This solution was found via the Stackoverflow website [51] and by meticulously sifting through Facebook's GitHub page for React. An excerpt of the code responsible for bypassing the whitelist with the *is* keyword can be seen in Appendix A.

The alternative approach would be to create functionality, adding X3DOM attributes to elements after they have been rendered by React. This is possible as React provides lifecycle events for components, in which attributes can be set programmatically on the X3DOM elements after they have been added to the DOM. This approach would circumvent the whitelist, but would slow down rendering performance as the DOM would have to be accessed directly, eliminating any performance gains provided by React's virtual DOM [49]. The virtual DOM in React refers to the representation of the UI that is maintained internally. It is used for staging changes to the DOM, which allows React to minimize slow, direct DOM manipulations.

To illustrate how X3DOM works with React (using the *is* keyword for disabling the whitelist), an example rendering of 5000 boxes was created. In *Code Listing 6* below, we can see the main component of this application written in JavaScript and JSX. The constructor of the component class generates 5000 positions and colors, adding them to the component's state. The render() method is called by React to get the rendering information from the component. This is where the JSX language extension for JavaScript is useful. It allows for writing HTML and React elements directly into a return statement (using parentheses if more than one line is written). This component returns the <*x3d*> element (the entry point of X3DOM), consisting of a scene containing all 5000 <*shape*> elements. The <*shape*> elements are added using the map function on the *position* array in the *state* object. The map function is a JavaScript function available to use on array types. It calls a "provided callback function once for each element of an array, in order, and constructs a new array from the results" [12]. This function behaves like a for loop in this application, with the important exception that it is created using a JavaScript expression. This enables it to be used inside the JSX code, as JSX allows any valid JavaScript expression in its syntax. For loops and if statements are control-flow constructs in JavaScript, and do not return a value [44]. This means that they are not JavaScript expressions, which is why the map function must be used for creating child elements dynamically instead. The callback function passed to the map function accepts the current index of the array in addition to the current element, which enables accessing both the *positions* and *colors* array from the state of the component. The respective elements from these arrays are used to declare a <*shape*> element with a specific position and color.

```
import React from 'react';
import { Shape } from './shape.js';

export class AppComponent extends React.Component {

    constructor() {
        super();
        let positions = [];
        let colors = [];
        for (let i = 0; i < 5000; i++) {
            positions.push(
                (Math.random() * 100 - 50) + ' ' +
                (Math.random() * 100 - 50) + ' ' +
                (Math.random() * 100 - 50));
            colors.push(
                (Math.random()) + ' ' +
                (Math.random()) + ' ' +
                (Math.random()));
        }
        this.state = {
            positions: positions,
            colors: colors
        };
    }

    render() {
        return (
            <div>
                <x3d width='640px' height='480px'>
                    <scene>
                        {
                            this.state.positions.map((x, i) => {
                                return (
                                    <Shape key={i.toString()}
                                        translation={x}
                                        color={this.state.colors[i]} />
                                );
                            })
                        }
                    </scene>
                </x3d>
            </div>
        );
    }
}
```

Code Listing 6: The React App component is responsible for the X3DOM scene in the example.

In *Code Listing 7* below, the code for the shape React component can be seen. The constructor in this component has a parameter called *props*. This parameter is where the attributes from the element declaration in a parent component is stored (ie. the color and translation attributes). The render method again uses JSX syntax to declare the component's view. The <*transform*> and <*material*> elements use the translation and color, respectively, to declare the position and color of the shape component.

One of the things that makes React difficult to use along with X3DOM, is that none of X3DOM's attributes are on React's attribute whitelist. The *is* attribute in JSX disables the whitelist for the element it is added to, and is therefore the only way we were able to make X3DOM work inside React components, however this is undocumented behaviour and may disappear in new React releases. The *is* keyword is shown in bold, underlined font in *Code Listing 7.*

In this example, the X3DOM shapes should listen for a click event, with the purpose of displaying more of X3DOM's behavior inside React. Since events from X3DOM nodes can only use listeners set up after X3DOM has processed the elements in the DOM, React's event bindings can't be used. Instead the React lifecycle event *componentDidMount,* which is triggered once the component's view is added to the DOM, can be used to add an event listener manually using DOM functionality. The *ref* attribute is used to make React add a reference to the <*transform*> element. In the process of rendering the component, React will add this reference to the *refs* property of the shape element. This eliminates the need to call *document.getElementById* for access to the element. The result of the rendering can be seen in *Figure 7* below.

```
import React from 'react';

export class Shape extends React.Component {
    constructor(props) {
        super();
        this.props = props;
    }

    click(e) {
        console.log(e);
    }

    render() {
        return (
            <transform is translation={this.props.translation} ref='comp'>
                <shape>
                    <appearance>
                        <material is diffuseColor={this.props.color}></material>
                    </appearance>
                    <box></box>
                </shape>
            </transform>
        );
    }

    componentDidMount() {
        this.refs.comp.onclick = (e) => this.click(e);
    }
}
```

Code Listing 7: React Shape component responsible for one box, used in the App component.

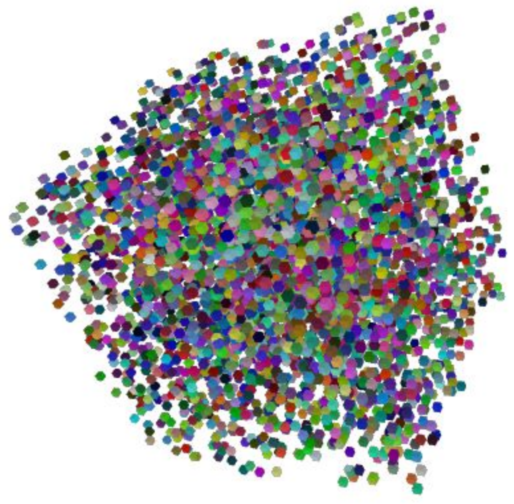


Figure 6: The result of the React application from Code Listing 6 and 7.

3.2.2 Angular

Angular only supports standard properties and a small set of attributes (mostly SVG related attributes) in its templates. Like React it also allows any element tag to be added to its view templates.

The main issue when using X3DOM inside a web framework that creates custom HTML elements in the form of components, is that non-X3DOM elements can't be used inside the X3DOM scene. This causes X3DOM to fail at reading the scene and produce an error, since X3DOM only understands its own syntax. Because of this, components created with a web framework, that do not have a name which match the name of an X3DOM node, cannot be used to wrap X3DOM elements inside the scene.

The solution to the issue mentioned in the previous paragraph, is using attribute selectors instead of element selectors on all components that are to be used inside the *<x3d>* element tag of the page. There are two ways to use an Angular component in another component's template. These are termed *element* and *attribute* selectors. Adding an Angular component to a template using an element selector means writing it as a traditional HTML element tag (e.g. *<my-component>*). An attribute selector works by adding the selector as an attribute to a regular element (e.g. *<div my-component>*). Which of these selectors that is used, is configured in each individual Angular component. This attribute selector must be added to an X3DOM *<transform>* element, since this element can be used to wrap other elements in X3DOM.

In addition to custom components, Angular has the concept of an attribute directive. An attribute directive in Angular is used for changing the appearance or behavior of an element or component (e.g. highlight when the mouse pointer hovers over a paragraph). By its name, the attribute directive is added to an element by declaring it as an attribute on that element (e.g. *<p highlight>*). Using attribute directives, one can add Angular support for the X3DOM attributes needed in an application by creating an attribute directive without functionality. Registering these directives with the proper attribute name from X3DOM as selectors will allow the use of corresponding X3DOM attributes in Angular.

To illustrate how X3DOM works with Angular, the rendering of boxes was also implemented in in Angular. *Code Listing 8* shows the App component, which is the entry point of the application. The *@Component* class decorator defines the selector (element name), and the template for the view of the component. The *<x3d>* and *<scene>* elements are declared, with a *<transform>* element inside the *<scene>*. The *<transform>* element has an attribute and some properties added to it. The **ngFor* attribute is Angular's way of creating a loop inside a template. The element containing the **ngFor* is added to the view *n* times (*n* being the number of iterations in the loop). The first part of the expression inside the quotes are mandatory, creating a local variable *pos* containing the current value of the *positions* array. It is also possible to create a local variable with the current loop index as its value. The *boxshape* attribute is the attribute selector for the BoxShape component, signaling to Angular that the component should be created inside this element. As mentioned previously, the <*transform*> element in X3DOM can be used to modify the translation of its child elements. The *pos* variable of the loop is bound to the *translation* attribute using Angular's attribute data binding [45]. Angular only allows known properties in data bindings, but binding to an attribute is possible if the attribute name is prefixed with "attr.". Lastly the local variable *i* is used to bind the *i*-th value of the *colors* array to the *color* property of the BoxShape component. In the constructor of the App component class, 5000 positions and colors are generated. These are added to the *positions* and *colors* arrays used in the template loop.

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  template: `
    <x3d width='640px' height='480px'>
      <scene>
          <transform
              *ngFor="let pos of positions; let i = index;"
              boxshape
              [attr.translation]="pos"
              [color]="colors[i]">
          </transform>
      </scene>
    </x3d>
  `
})
export class AppComponent {

  positions:Array<string> = new Array<string>();
  colors:Array<string> = new Array<string>();

  constructor() {
    for(let i = 0; i < 5000; i++) {
      let pos = (Math.random()*100-50) + ' '
              + (Math.random()*100-50) + ' '
              + (Math.random()*100-50);
      let color = Math.random() + ' '
                + Math.random() + ' '
                + Math.random();
      this.positions.push(pos);
      this.colors.push(color);
    }
  }
}
```
Code Listing 8: The App component of the Angular version for rendering the boxes.

Code Listing 9 shows the code for the BoxShape component. The selector in this component is different from the App component. It is an attribute selector declared with brackets surrounding the selector name. This lets the selector be used directly on elements as an attribute, and is thus named an attribute selector. It is important for the integration with X3DOM that the components can be added to an element, since X3DOM will throw an error if a non-X3DOM element is added inside the enclosing <*x3d*> element. The attribute selector solves this problem since it can be added directly onto the X3DOM <*transform*> element. In the template there is another attribute data binding, setting X3DOM's *diffuseColor* to the value of the *color* variable of the class. The *color* variable is defined as an input to the component. The constructor of the BoxShape component contains a parameter of type *ElementRef* [46], and adds this as an instance variable of the class. What happens behind the scene here, is that Angular's Dependency Injection mechanism injects the *ElementRef* service into the component, which contains a reference to the component's native element in the DOM when it has been rendered.
The *ngAfterViewInit* event is one of Angular's lifecycle events, similar to React's *componentDidMount*. The same workaround that was performed in the React example for listeners on events from the X3DOM components must be used in Angular also. The listener function is set up when the component has been rendered. A listener for the DOM's *load* event is added to be certain X3DOM has loaded and replaced the events on the elements in the scene. The *elementRef* variable is then used to access the element, and add an event listener for the click event. The result of the rendering can be seen in *Figure 7* below.

```
import { Component, Input, ElementRef, AfterViewInit } from '@angular/core';

@Component({
  selector: '[boxshape]',
  template: `
    <shape>
      <appearance>
        <material [attr.diffuseColor]='color'></material>
      </appearance>
      <box></box>
    </shape>
  `
})
export class BoxShapeComponent implements AfterViewInit {

  @Input('color') color;

  constructor(private elementRef:ElementRef) { }

  ngAfterViewInit() {
    document.addEventListener('load', () => {
      this.elementRef.nativeElement.addEventListener('click', (event) => {
        console.log(event);
      });
    });
  }
}
```

Code Listing 9: The BoxShape component of the Angular version of the boxes example.

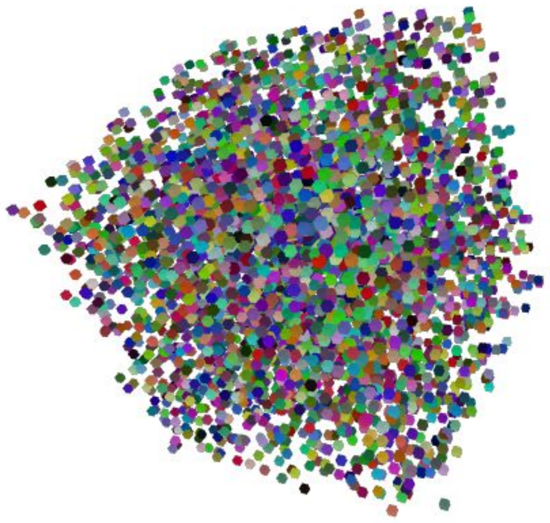


Figure 7: The result of the Angular application from Code Listing 8 and 9.

3.2.3 Choice of web framework

While both Angular and React were able to create the simple example in the previous section, one key difference was apparent. React can't use attribute selectors for inserting elements into the DOM, instead relying on the JavaScript class name being used as an element selector. X3DOM requires each element tag inside the X3D element root to be a known X3DOM node name. Since this means that all React components used inside the X3D element root must have a name that is also the name of an X3DOM node, the number of components that can be wrapped by React is limited to the number of X3DOM nodes. This also means that not more than one component can wrap a specific X3DOM node, to provide alternative functionality. Accordingly, we were not able to compose more advanced components using previously defined components, since the required class names will already have been used. Because of this, a method to use React for the development of the application was not found.

In contrast, Angular allows what it terms as attribute selectors, which lets components be used by adding an attribute to an element, enabling the creation of an unlimited number of different integrated X3DOM nodes. The only requirement is that X3DOM properties are defined as Angular directives so that Angular is aware of them (otherwise an error is thrown), and allows them to be added to elements in templates.

Given these approaches to integrating X3DOM with each respective web framework, Angular was chosen for further investigation, while React was discarded. This was due to Angular appearing to have the least amount of obstacles in the integration path. React did have the option of disabling the attribute whitelist, but since this was not a documented feature it was not deemed stable, and therefore not a viable option. Angular's approach was more straightforward than adding attributes after render time in React.

4 Problem solution

As described in section 1.1, the target application should visualize data at correct geospatial coordinates, including depth, in a scene containing topography of the area surrounding the data. It is important that the visualizations can easily be toggled on and off, since they can overlap. Publications and reports that are relevant to individual visualizations should be easily accessible through interactions with that visualization. The implementation of this application is described in the following sections, divided into three parts. They are the rendering of terrain (topography), visualization of geological slice data and visualization of well data.

4.1 Visualization of terrain

For our application, a central part is to visualize terrain for creating a context to where the subsurface data resides. Terrains can cover large areas and contain a large amount of triangles. To achieve interactivity, a level-of-detail (LOD) scheme is often applied for terrain visualization. A LOD scheme achieves interactivity by showing coarser geometry for terrain far away from the viewpoint and finer geometry for terrain close to the viewpoint. Doing this creates an upper bound on how many triangles are rendered. Terrain LOD algorithms can be complex to implement, but luckily there is a ready implementation in X3DOM called the BVHRefiner node. The BVHRefiner node creates a terrain with LOD functionality. The terrain is constructed from heightmap images of an area. In the application, these are statically added to the public folder of the server, so that X3DOM can access them via a configured base URL. The application does not support user uploads of additional areas. The WMTS [17] addressing scheme is used for fetching the correct images for the current LOD of the BVHRefiner from the public folder. The BVHRefiner requires images to be stored using this scheme when a three-dimensional terrain is rendered. The WMTS addressing scheme consists of one folder per level of detail, which contains a number of images the BVHRefiner can use as tiles to create the terrain with the appropriate LOD. Each folder is named by their level, containing a matrix of tiles for that level. Each LOD folder contains as many folders as there are columns in that level of detail, and each of these folders contain as many images as there are columns in the current LOD folder. An overview of the WMTS directory structure can be seen in *Figure 8* below [17].

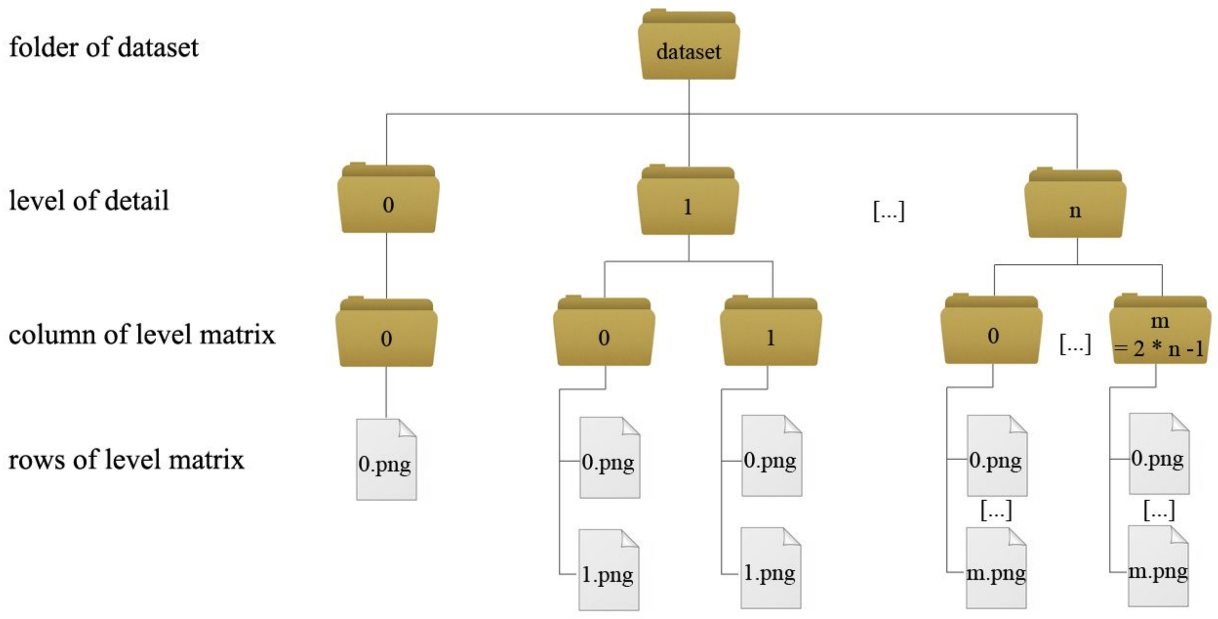


Figure 8: An overview of the WMTS directory structure. The figure is copied from X3DOM's documentation [17].

The original height-map image is extracted from a GeoTIFF file of Svalbard, fetched from the Norwegian Polar Institute [18]. This is accomplished using a tool called GDAL (Geospatial Data Abstraction Library), which is a set of programs and libraries providing access to geospatial data in raster or vector formats, and tools for exporting specific information from geospatial data into different formats [19]. In addition to extracting the heightmap, GDAL was used in this thesis to generate a hillshade image as a texture for the terrain. The BVHRefiner node can apply LOD textures to the terrain it creates using the LOD heightmap images.

The BVHRefiner node has a number of attributes available for configuration. These can be seen in *Code Listing 10*. The *maxDepth* is the maximum depth, and the *minDepth* is the minimum depth in the LOD tree. The *interactionDepth* is the LOD depth that will be used when interacting with the scene, e.g. when moving. The *subdivision* is the resolution of a rendered tile of the plane, while the *size* is the size of the entire terrain plane. The *factor* is used when determining the distance the next level of detail will be rendered at. The lower it is, the higher the quality of the rendering. The *maxElevation* affects the scaling along the Y-axis. The *elevationUrl, textureUrl* and *normalUrl* specify the URLs to the datasets defining height values, textures and normals, respectively, for the BVHRefiner. The *elevationFormat*, *textureFormat* and *normalFormat* specify the file format of the images in each dataset. The *mode* property is the different display modes supported. This thesis only uses the 3D mode. The *submode* is the addressing scheme for the LOD datasets. Only the WMTS addressing scheme is supported in 3D mode.

```
<BVHRefiner maxDepth='5'
            minDepth='2'
            interactionDepth='2'
            subdivision='64 64'
            size='4096 4096'
            factor='10'
            maxElevation='410'
            elevationUrl="/elevation"
            textureUrl="/texture"
            normalUrl="/normal"
            elevationFormat='png'
            textureFormat='png'
            normalFormat='png'
            mode="3d"
            submode="wmts">
       </BVHRefiner>
```

Code Listing 10: The BVHRefiner node, declared with its required attributes.

The attributes *maxDepth*, *minDepth*, *interactionDepth* and *factor* can be changed interactively by the user to increase detail or performance. This allows devices with lower graphics performance to also make use of the application. In *Figure 9* below, the topography rendering of the finished application can be seen.

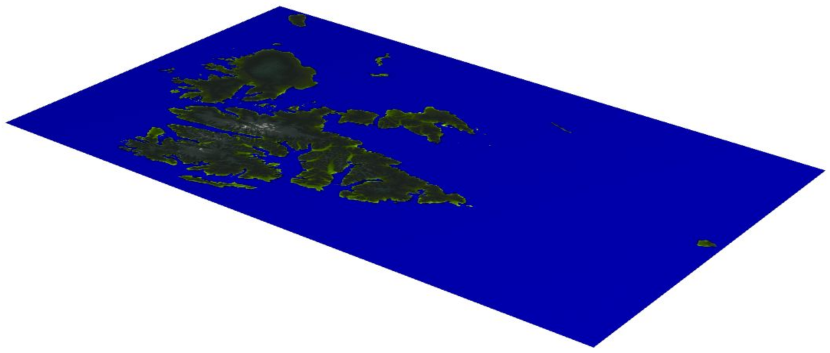


Figure 9: Bird's eye view of the topography of Svalbard, as rendered by the X3DOM BVHRefiner node.

4.2 Visualization of 2D geological data

4.2.1 Slice module

The largest module of the application is the slice module. This module controls the loading of, visualization of, and interaction with the two-dimensional geological data. The data is provided as images, displaying a slice of the geology at specific coordinates. In *Figure 10* below, a diagram of the architecture of the slice module can be seen. The visualization of slices happens in the Slice component. It uses the Slice service for fetching the slices to visualize, then generates X3DOM code for each slice, which it finally inserts into its view using standard DOM functionality. The storage and state of the slices rendered in the scene are managed by an Angular Service [21], Slice service. The service loads slices from the server, stores these in an array of Slice data types, and makes them available to the rest of the slice module.

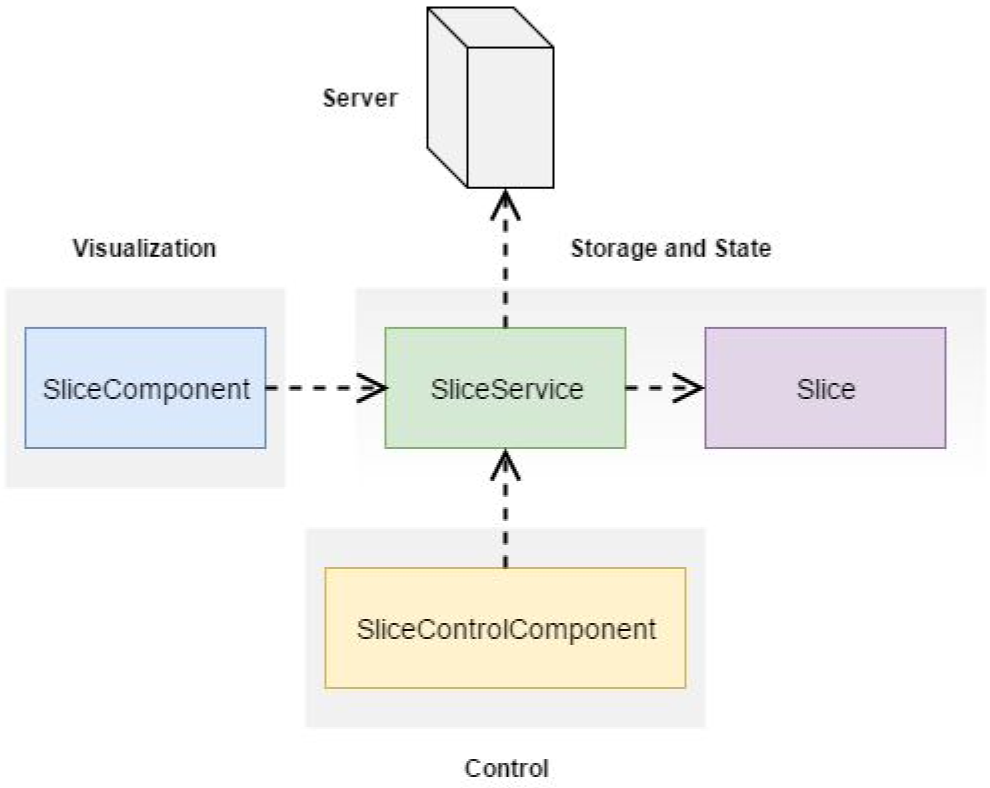


Figure 10: The architecture of the slice module.

Changing the visualization of slices can be done by interacting with the visualization itself, or by using the SliceControl component. This component allows operations such as toggling individual slices and navigating to each slice in the scene. It also uses the Slice service to notify the rest of the application of changes. This approach creates a decoupled architecture where the Slice service manages all interactions between components.

4.2.2 Storage and state

Storage and state in the Angular framework should not be handled in components. Angular components should be lean, and only concern themselves with the user experience. The one-fits-all for tasks that do not belong in a component, is a service. Components are consumers of services and should use these for, among others, fetching and storing data. A service is not a special type defined in Angular, but a standard class with methods for performing the intended task(s) [21]. This class is registered with an Angular module, which allows Angular's dependency injection mechanisms [20] to provide instances of the service to components which depends on it.

The dependency injection system (DI) in Angular is responsible for injecting required services into components that are dependent on said service. The DI maintains a container with instances of services. When creating an instance of a component, the framework identifies the services needed, and looks for an instantiated object in the service container. The instance is then injected into the component, or in the event that there is no instance in the container, an instance is created and added to the container before it is injected [20]. The fact that Angular manages the creation of services in this manner makes them well suited for storing data that needs to be shared between components, as the data will be available to all components without any additional configuration required by the developer. Another benefit is that when a service is instantiated at the root-level component, and several components require the same data from a server, the number of HTTP requests can be reduced to only a single one when data is stored in the service itself. Provided that the data fetched from the server is not expected to change often, the service can return the stored data if the HTTP request has already been performed by another component. It is possible to tell Angular to create a new instance of a service for a component. This service instance will then be injected into that component and all children that use the service. This is done by adding the *provider* property to the object passed to the *@Component* decorator in the component class, and declaring an array containing the service as the value of the property.

4.2.2.1 Slice

It is considered good practice in Angular to create data types, since TypeScript is a statically typed language, and can perform compile-time type checking. This can eliminate common errors when the program is compiled instead of the program failing during runtime, thus speeding up development. The Slice class is such a defined data type, and contains the properties that a slice consist of.

4.2.2.2 Slice service

Angular services are used to create all the functionality that should not be in the component itself. The Slice service handles the calling of the server API, fetching the slice data residing in the database. Angular's built in HTTP service is used to accomplish this. The HTTP service contains functionality for communicating over the HTTP protocol. It simplifies the task of using AJAX to access resources from a server [22].

When using the HTTP service to perform an HTTP request, the service returns an RxJS Observable. RxJS is the JavaScript version of ReactiveX, a library for "reactive programming using Observables, to make it easier to compose asynchronous or callback-based code" [23]. The Angular team endorses RxJS, and Observables are used extensively in Angular applications [24]. An Observable is an object that emits one or a set of events that can be subscribed to by an Observer, and subsequently processed by the Observer using a multitude of available operators. These operators are similar to array operators in JavaScript and TypeScript (for instance, the *map* function mentioned in chapter 3 is an available operator). *Figure 11* below shows the data flow of an Observable. An entity responsible for the Observable (in this case the HTTP service) calls the *onNext* method of the Observable, letting the Observable know that new data is available so it can notify its Observers. An optional step that can be performed on the Observable in the controlling entity or in an intermediary function, is Observable operators. These can be chained and combined, allowing for transformation and manipulation of the items emitted from the Observable. After all the operations are completed, the subscriber is notified with the value of the Observable [25].

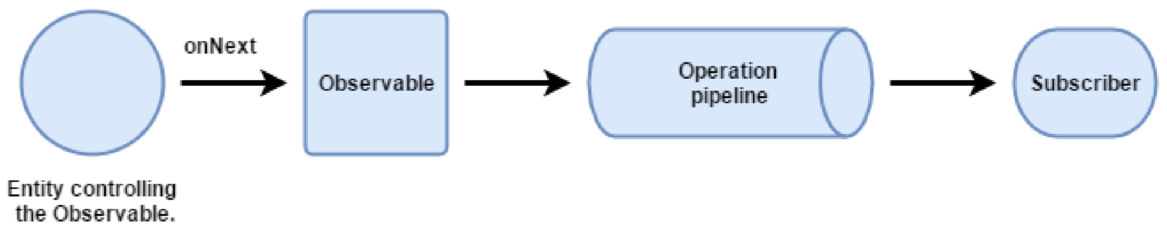


Figure 11: The data flow of an RxJS Observable.

In *Code Listing 11* below, Slice service's method for loading slice data can be seen. Angular's HTTP service is injected into the instance variable *http* of the class during instantiation of the class. It is used here to perform a GET request to the URL specified. As mentioned, the HTTP service returns an Observable. The *map* operation is performed on the Observable, and the

Observable is returned from the *loadSlices* method. The *map* operation converts the response data to a JSON object, and returns the new result to the Observable.

```
loadSlices() {
    return this.http.get(this.sliceResourceUrl)
        .map((res: Response) => {
            let body = res.json();
            return body || {};
        }).catch((err: any) => {
            return err;
        });
}
```

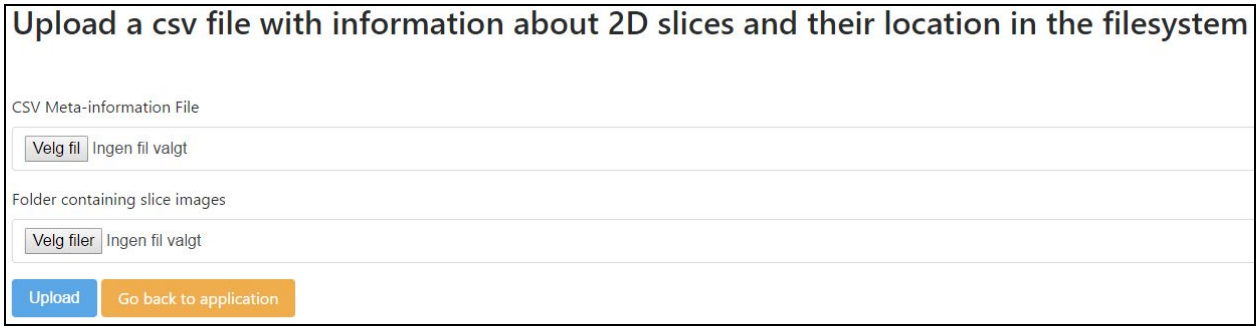
Code Listing 11: The Slice service's loadSlices method, using Angular's HTTP service for performing a GET request.

Another RxJS type is a Subject [26]. A Subject implements both the IObserver and IObservable interfaces of RxJS, and can therefore act as both an Observer and Observable. Additional functionality for, among others, emitting items to Observers using the Subject's *onNext* method is available. In fact you can control the entire data flow with a Subject, using the *onNext, onError,* and *onCompleted* methods. When a Subject's *onNext* method is called, the Subject will notify all Observers of the Subject with the item passed to the *onNext* method. The ReplaySubject class [26] is a derived class of Subject that instead of emitting just the item passed to *onNext,* emits all the items that the ReplaySubject has received. This is useful for when the *onNext* method is called before Observers subscribe to the Observable, as this lets the new Observers receive the items also. The ReplaySubject can be created with a specific size of the buffer keeping track of old items, to limit the amount of items that new subscribers are notified of when they subscribe [26].

Since the Slice service is responsible for managing all the slices in the application, it calls the *loadSlices* method itself when instantiated. The slices are stored in an instance variable of the Slice service. The service holds a ReplaySubject that components which need to receive the slices can subscribe to. The *onNext* method of the ReplaySubject is called after the slices have been loaded. The Slice service uses the ReplaySubject since the service might load the slices before the components that need them are instantiated. This lets those components subscribe to the service's ReplaySubject, and receive the slices even if they are instantiated too late to receive the initial event from the Subject. The reason this approach to service design was used is that it provides a simple way to ensure the slices are only loaded once, while also enabling a simple method for components to receive updates to the slice collection stored in the service.In addition to the one ReplaySubject that notifies about all slices from the server, the slice service has a ReplaySubject for emitting a single slice. Subscribers to this subject will be notified when a single slice is changed. The slice service contains a method for changing a single slice, which will then trigger both the subject for single and all slices. This is useful for eliminating the need for components using the slice data to iterate over the entire array of slices to update their views.

4.2.2.3 Uploading slices to the application

The geological slice data is basically two-dimensional images with position and extent. To upload them, the images must be uploaded, and UTM coordinates and extent information such as depth is needed to correctly visualize the slice in the scene. Meta information about the slice in the form of name of the dataset, a description and list of links to related literature is also uploaded to give the user of the application background information about a slice. This was provided by CMR, with slices represented as rows in a Microsoft Excel spreadsheet document, along with the path to the image files. The rows of the spreadsheet contained extent, positional and meta information about the slices.


Upload a csv file with information about 2D slices and their location in the filesystem

CSV Meta-information File

Velg fil Ingen fil valgt

Folder containing slice images

Velg filer Ingen fil valgt

Figure 12: A screenshot of the page for uploading slices.

Since CMR stored the necessary information for uploading slices in Microsoft Excel spreadsheets, the uploading of slices needed to happen in bulk, by uploading the spreadsheet. A file dialog where the user can browse the filesystem and select the file was used for this. The only issue with this approach is that web browsers cannot open files on a host system without user interaction, even though the file paths were specified in the spreadsheet. This was solved by uploading the images using a separate file dialog configured to permit multiple files. This allows users to upload all relevant images alongside the spreadsheet. The upload code then matches the image filenames with the names in the spreadsheet, and creates slice datatype objects to upload to the server. A JavaScript library called PapaParse [27] was used for parsing the spreadsheet. PapaParse reads a spreadsheet file, and creates an array of JavaScript objects representing the rows in the file [27]. Note that it is not possible to use Excel files with this library, they will have to be exported to CSV files before upload. When the spreadsheet file and the images both have been uploaded using the file dialogs, the user can press the upload button. The application then loops over the list of JavaScript objects generated by PapaParse, and matches the objects to the images. When a match is found, the meta information from the JavaScript object is processed for upload to the server. The UTM coordinates are transformed to latitude and longitude, and are then mapped to the local coordinate system of the application. The image file is added to the the JavaScript object, and uploaded to the server. *Figure 12* above displays the upload page for slices.

4.2.3 Slice visualization

The visualization logic of the slice module resides in an Angular component, but the logic itself is plain TypeScript and JQuery [28]. JQuery is a library for JavaScript (that works in TypeScript also), containing functionality that makes it easier to, among others, traverse and manipulate the DOM, and handle events. JQuery is used by the application because of this functionality, and particularly because of the library's methods for easy creation, addition and removal of nodes from the DOM [28].

The Slice component's view template is a single X3DOM <*transform*> element, containing an *id* attribute. This id is used to get a reference to the element using JQuery, allowing the component's methods to add X3DOM elements inside the <*transform*> element during runtime. The component uses the Slice service to fetch the slices to be visualized. It also listens for changes to individual slices in the Slice service. When the slice data has been loaded, the component processes each slice and generates X3DOM code describing the visualization. Click event listeners are created on the <*shape*> X3DOM elements in the X3DOM code. The individual pieces of X3DOM code for each slice are then concatenated, and inserted into the <*transform*> element of the component's view. Since the X3DOM code is added to an element in the component's view using standard DOM functionality instead of Angular, the component itself needs to be added to the DOM first. To ensure that this happens, the entry point of the internal TypeScript logic is the *AfterViewInit* lifecycle function of the component, as this function is called by Angular after the component has been added to the DOM. Below follows a more in-depth explanation of the generation of X3DOM code used in the Slice component.

The visualization of data in the application has some requirements, as mentioned in section 1.1. Visualized data must be placed at the correct position in the scene, as specified in the data file. It must also be possible to toggle the visualization of the data by interacting directly with the visualization, along with accessing additional information about the data.

The toggle functionality requires a substitute visualization to be shown in the slice's place when it is toggled off, indicating that there is hidden data at this position. The slice can then be toggled back on by interacting with the substitute visualization. The substitute visualization used in the application is a red, translucent sphere. Whether the slice shape or the substitute shape is displayed, depends on the value of the *visible* property of the slice data object. Both of these shapes can be seen in *Figure 13*. Because of the requirement of toggle functionality, the Slice component has a function for generating the X3DOM code for both the slice and the substitute shape for a given slice. The function that creates the X3DOM code for a slice can be seen in *Code Listing 12* below. The <*transform*> element is used to create a local coordinate system in X3DOM, with the *translation* and *rotation* describing how the local coordinate system is translated and rotated in its parent coordinate system. The slice data has both a translation and rotation field intended to be used for this purpose, and these are used as values on the element. The <*shape*> element and its children declares the appearance of the slice shape. A texture is added to the appearance, using the texture URL of the slice object. The geometry of the slice shape is declared as a box, where the dimensions of the box are specified by the slice's *size* property. The template for a slice also includes an X3DOM <*viewpoint*> element. This element lets developers specify a location and orientation that can be applied to the camera, so that the camera can be quickly moved to a point of interest in the scene. X3DOM allows for imperatively setting a viewpoint in the scene as active. When this is done, the camera navigates to the viewpoint with an animation to ensure a smooth transition of the view. This navigation functionality is further explained in section 4.2.4.

```
getSliceTemplate(slice):string {
       return `
           <transform
               id="${slice.id+'_outer'}"
               translation="${slice.translation}"
               rotation="${slice.rotation}">

                   <shape id="${slice.id+'_slice'}">
                       <appearance>
                           <ImageTexture url="${slice.textureUrl}"></ImageTexture>

                       </appearance>
                       <box size="${slice.size}"></box>
                   </shape>

                   <Viewpoint
                       id="${slice.id+'_viewpoint'}"
                       position="0 -150 0"
                       orientation="-30 1 0 -1.57">
                   </Viewpoint>
           </transform>
       `;
   }
```

Code Listing 12: The function that generates the X3DOM code for a single slice, as found in the Slice component.

The component also has an almost identical function as the one in *Code Listing 12*, that generates the X3DOM code for substitute shapes. Here the geometry of the substitute shape is instead declared to be a sphere, and the texture used is a semi-transparent red texture. The functions for generating X3DOM code for slices and substitute shapes are used by the component during setup of the X3DOM view.

When all the X3DOM code has been added to the DOM, a new loop over the slices adds an event listener for the click event to each slice or substitute shape, so that the application can handle user interactions with the visualizations. The event listener function the slices differentiates between left and right clicks. If a left click is registered, the visibility of the slice is inverted. The X3DOM code for the clicked slice or substitute shape is then removed from the DOM, and replaced with the X3DOM code for the opposite shape. X3DOM sees the change in the DOM, and updates the scene accordingly.

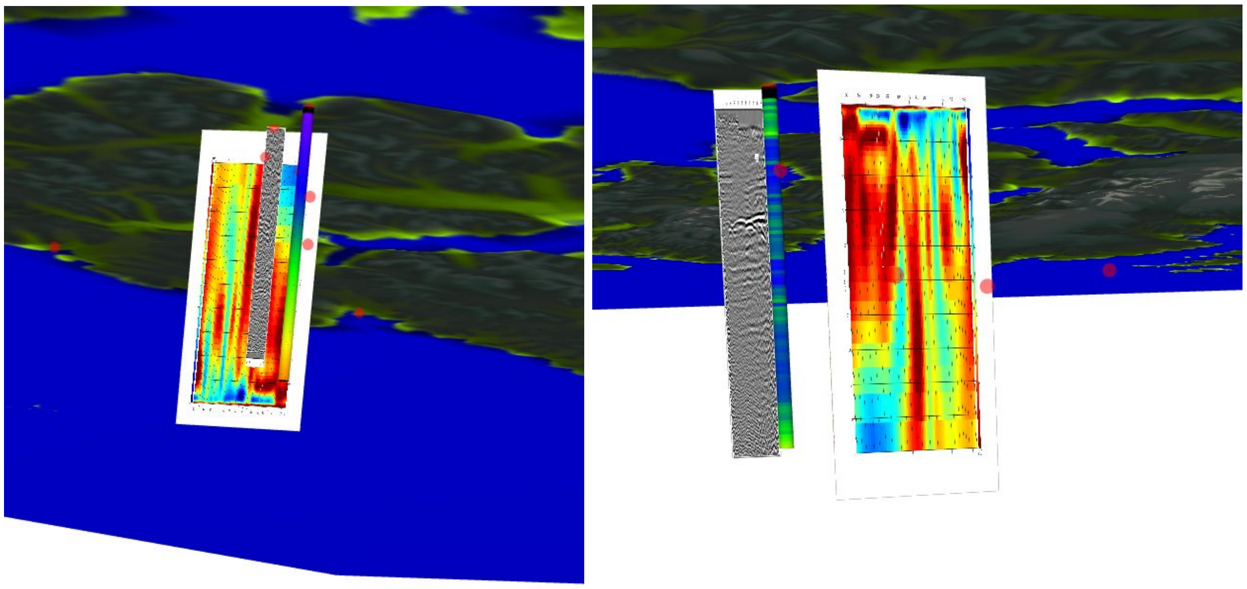


Figure 13: Two images of the application visualizing data. Seen from below the surface, a resistivity (colored) and seismic (grayscale) slice can be seen in the center of the images. The red spheres are the substitute shapes for slices that have been toggled off. There is also a well visualization (blue-green vertical cylinder), which is further described in section 4.4.

4.2.4 Slice visualization control

In addition to the direct interactions with the visualizations described in the previous section, control of visualizations using the SliceControl component in the sidebar menu of the application is supported.

The SliceControl component is a UI component for controlling the slice visualization. The UI of the component consists of a list, with elements of the list representing slices. Each element of the list contains a button for toggling a slice, and for navigating to the viewpoint of a slice. The list elements also have a button that allows users to open the information modal window for a slice. The SliceControl component subscribes to the Slice service to access the global list of slices, which is used to populate the control list. The Slice service is also used to notify the Slice component of the toggling of a slice (so it can update the view), and to ensure that the control list is updated when a slice is toggled using direct interaction (i.e. clicking). The application flow from user interaction to visualization change can be seen in *Figure 14*. The figure shows that both direct interactions via clicking, and interactions via the SliceControl component can change the visualization. Both options use the Slice service to notify all relevant components of the change. Only the SliceControl component can be used to navigate to a slice. In the Slice component, the *id* attribute of the <*viewpoint*> element declared in the generated code for a slice is set to the *id* property of the slice (see *Code Listing 12)*. This allows the view button in list items to move the camera to the viewpoint of a slice by using DOM functionality to access and bind the correct viewpoint.

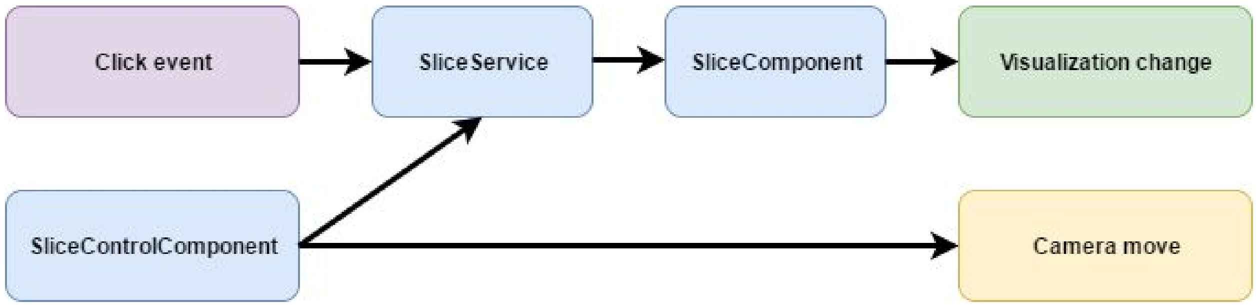
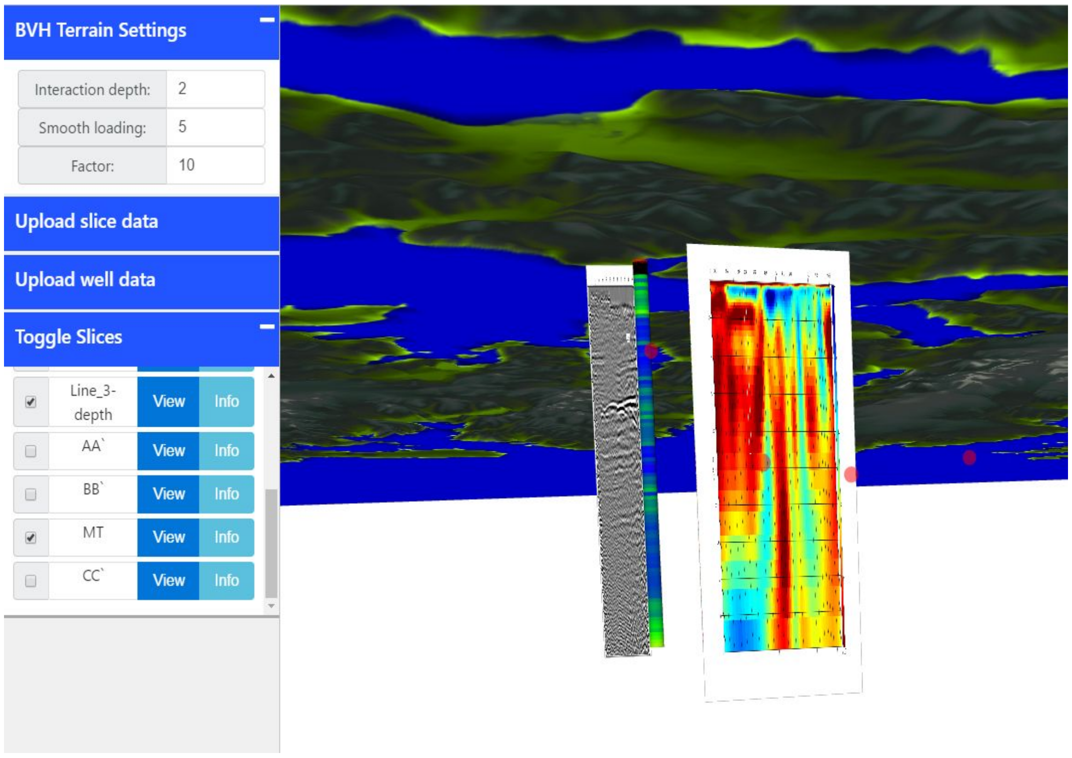


Figure 14: Diagram of the application flow from user interaction to visualization.

The UI of the SliceControl component can be seen in *Figure 15*. There is a list item for each slice in the scene. The name of the slice is displayed, along with control buttons. The checkbox toggles the *visible* property of the of the slice on and off. The view button in the list items moves the camera to the viewpoint for a slice. The info button opens the information modal window for a slice.



Interaction depth: 2
Smooth loading: 5
Factor: 10

✓	Line_3-depth	View	Info
	AA`	View	Info
	BB`	View	Info
✓	MT	View	Info
	CC`	View	Info

Figure 15: The slice control (Toggle Slices) menu expanded in the sidebar of the application. A list of the slices rendered in the scene can be seen here, along with a checkbox for toggling, the slice name, and buttons for navigating to the slice and viewing the information window of the slice.

4.2.5 Angular issues using X3DOM inside templates

The visualization of slices was implemented in two different ways, as was the visualization of wells. First with plain Angular components, as described above, and then with regular TypeScript/JavaScript. This was because Angular turned out not to work completely with X3DOM. The implementation in Angular is described briefly below, followed by information about why the use of Angular components containing X3DOM in their view template failed in the implementation.

4.2.5.1 Slices

The visualization of the geological slices was done using three Angular components: The SliceGroup component, Slice2D component, and BoxShape component. The relationship between these components can be seen below in *Figure 16*. The SliceGroup component is responsible for creating Slice2D components, which control the logic of one slice. This component performs the necessary calculations for adding a slice to the scene, and creates a BoxShape component that renders the slice's geometry and specifies its appearance.

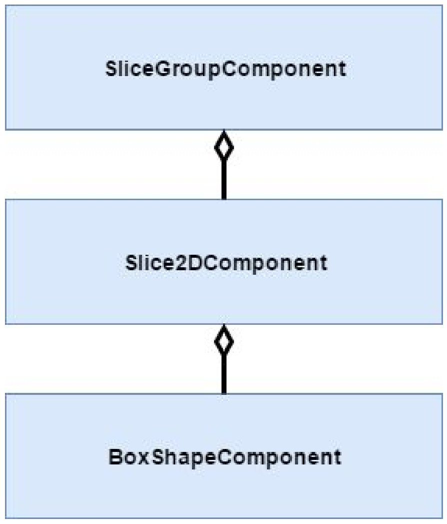


Figure 16: Diagram of the visualization components' relationship in the initial version.

4.2.5.2 SliceGroupComponent

The SliceGroup component fetches the slice data objects to be visualized from the Slice service, and creates Slice2D components for each slice returned by the service, passing a slice data object as a property to this component. In addition to creating Slice2D components, a viewpoint node is added along with each slice.

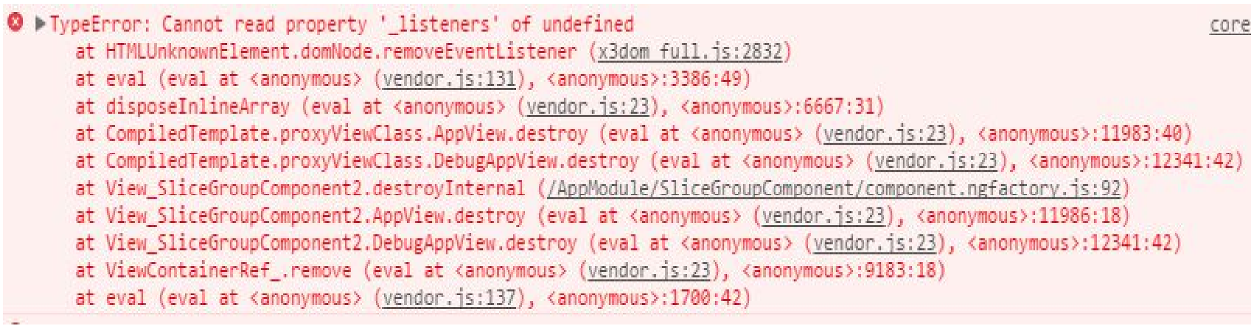
4.2.5.3 Slice2DComponent

The Slice2D component controls the behavior and display of a single slice. The view logic of the Slice2D component contains two components: The BoxShape component, and the SubstituteGeometry component. The BoxShape component is an Angular-wrapped X3DOM <*shape*> node, declared with a box geometry. It receives a texture URL and a size as input properties from the Slice2D component, and uses these to add texture and specify the size of the box, respectively.

The SubstituteGeometry component is an Angular-wrapped <*shape*> node just like the BoxShape component, but uses a sphere geometry, and a single color instead of a texture. Only one of the BoxShape and the SubstituteGeometry components is visible at any time, controlled by the slice data's *visible* property in the view logic. If the slice is toggled off, the SubstituteGeometry component is displayed. Otherwise, the BoxShape with the slice texture is displayed.

4.2.5.4 Application crash

When the toggling of slices was implemented, issues with the integration between Angular and X3DOM appeared. X3DOM's events work with the regular DOM functionality of the browser by replacing native event functionality with its own. The application waited for the Angular components to finish loading before event listeners were added, so that these replaced *addEventListener* functions were used. However, X3DOM also replaces the *removeEventListener* functions, and Angular use these to remove any listeners on element when the view of a component is destructed. No errors regarding this appeared initially, but late in the development work, the error seen in *Figure 17* occurred. The error was observed at different occasions, but most frequently when the slice shape was toggled on or off. In *Figure 16*, the issue is caused when the Angular framework attempts to remove event listeners from an element in a component's view when it is being destructed.



```
▶TypeError: Cannot read property '_listeners' of undefined                                                                    core
    at HTMLUnknownElement.domNode.removeEventListener (x3dom_full.js:2832)
    at eval (eval at <anonymous> (vendor.js:131), <anonymous>:3386:49)
    at disposeInlineArray (eval at <anonymous> (vendor.js:23), <anonymous>:6667:31)
    at CompiledTemplate.proxyViewClass.AppView.destroy (eval at <anonymous> (vendor.js:23), <anonymous>:11983:40)
    at CompiledTemplate.proxyViewClass.DebugAppView.destroy (eval at <anonymous> (vendor.js:23), <anonymous>:12341:42)
    at View_SliceGroupComponent2.destroyInternal (/AppModule/SliceGroupComponent/component.ngfactory.js:92)
    at View_SliceGroupComponent2.AppView.destroy (eval at <anonymous> (vendor.js:23), <anonymous>:11986:18)
    at View_SliceGroupComponent2.DebugAppView.destroy (eval at <anonymous> (vendor.js:23), <anonymous>:12341:42)
    at ViewContainerRef_.remove (eval at <anonymous> (vendor.js:23), <anonymous>:9183:18)
    at eval (eval at <anonymous> (vendor.js:137), <anonymous>:1700:42)
```

Figure 17: The error message written to the console log of the web browser.

The line that causes the error originates from X3DOM, since X3DOM has replaced the *removeEventListener* function, and it can be seen below in *Code Listing 13*. The *this* context in the code is an element, and if it is an element that X3DOM has processed and added its own event logic to, it will have a property called *_x3domNode*. The error is caused by this property not existing on the element that was supposed to have its event listeners removed.

```
...
var list = this._x3domNode._listeners[type];
...
```

Code Listing 13: The line of code from X3DOM where the error occurs.

It appears that a component is destructed by Angular at a point when it has not been processed by X3DOM yet. This might happen if the input properties to an Angular component is changed before X3DOM has finished processing the elements in its view. The component's view would then be destructed and recreated, but during the destruction X3DOM functions would be called on an incomplete X3DOM element. Another possibility is that Angular could remove property values on elements before the *removeEventListener* function is called. This would also cause the error above.

After attempting to use a variety of delay functions and event listeners for different load events in the DOM, the attempt to integrate X3DOM code in Angular component templates was abandoned to ensure the application could be finished in the allotted time. The final architecture of the visualization part of the application is described in section 4.2.3.

4.3 Visualization of well data

4.3.1 Well module

The well module controls the loading of wells, and the visualization of the wells in the scene. In *Figure 18* below, the architecture of the well module can be seen. The architecture is to a large extent identical to that of the slice module. The Well component is responsible for the visualization of the wells. The Well service is responsible for loading the wells from the server, holding the state of the wells and holding the options for the visualization of the wells. The WellControl component controls the visualization of the wells, allowing the user to change what property of the well is being visualized at any time. There are more options for controlling the visualization of the wells than the slices, requiring a more advanced control component. The well control is accessed inside the information modal window of each well.

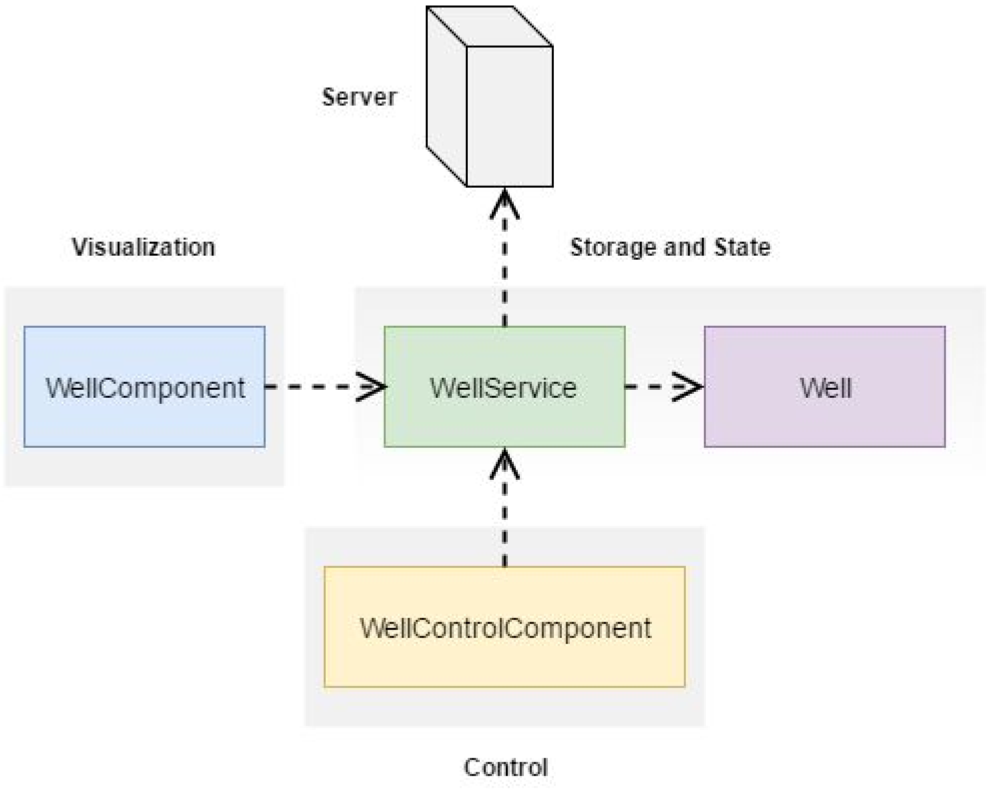


Figure 18: The architecture of the well module.

4.3.2 Storage and State

The well module has the same needs for storing the state of wells as the slice module has for slices. Visualization options for the wells are added to the well objects themselves when they are loaded from the server, ensuring that components using them access the same visualization state.

4.3.2.1 Well

Well data fetched from the server are represented on the client by the Well class. The class is a simple TypeScript class that holds all the properties of a well. A well contains an array of positions (with x, y and z values), where each position represents the start of a cylinder segment of the well, and the next position represents the end of the current segment and the start of the next one. An equally long array contains the values of properties on the well, with the names of the properties defined in yet another array. Examples of properties of a well can be pressure, temperature and permeability. The array of property values is organized by geospatial depth, and the property values for a specific well segment are accessed using that segment's depth value.

4.3.2.2 Well service

The well service fetches wells from the server when it is initialized, and stores them in an array of well objects as an instance variable. The service has two RxJS ReplaySubjects that are used for communicating with its observers. ReplaySubjects are as mentioned an RxJS Subject that keeps the last *n* items that have been passed to it to every new observer. In the same way as the slice service, the well service has a ReplaySubject that notifies observers of the list of wells fetched from the server. The other subject is used for notifying observers of changes to individual wells.

4.3.2.3 Uploading wells to the application

Uploading well data to the application requires more processing than with the slice data. The well objects are defined in RMS files, which are textual well log files. The well log contains information about different properties related to the surrounding geology and the drilling operation of the well, at different depths. Properties such as pressure and temperature are logged, and can be found in the log file, organized by position and depth.
As with the geological slices, the data for the wells provided by CMR was defined in a Microsoft Excel spreadsheet, though with accompanying RMS files instead of images. Accordingly, uploading the wells in bulk should be made possible for users by allowing upload of the spreadsheet. The spreadsheet contains much of the same information for wells as it did for slices, e.g. well name, description and related articles. Each well is represented by a row in the spreadsheet, and each row has a column with the name of the RMS file for the well.
The spreadsheet and RMS files are uploaded using the same method as for slices, described in section 4.2.2.3.

The RMS files provided by CMR are plain text files where the lines contain information organized as follows:

- Name of the dataset
- (X) Number of well properties in the file
- X lines containing the name of each well property
- From here until the end of the file follows lines with X + 3 space-separated columns. The first two columns contain the UTM coordinates (East, North), the third column contains the depth in meters. The remaining X columns contain the measured values of the well properties for this depth of the well.

Code for parsing the RMS files were added to the upload component for wells. The file is parsed by first reading the number of properties (X) in the well log. The following X lines are read into an array of property names. The remaining lines of the file are read one by one, splitting each line on white spaces, and creating a new array of the elements. The first two elements are the UTM coordinates of the log point, and are mapped into the local coordinate system of the application. The third element is the depth in meters of the log point, and is mapped in the same manner. Each of the remaining elements are the values of the properties of the well at the log point, and they are added to an array containing values from that point.
A new well object is created from the meta information of the spreadsheet and the parsed RMS file, and is uploaded to the server.

4.3.3 Well visualization

The visualization of the wells are done in much the same way as the slices. An Angular component is used, where its template contains only a single X3DOM *<transform>* element with an *id* attribute added so it is reachable by the DOM during runtime. As with the slice visualization, there is a substitute shape for the well in addition to the well shape, where which shape is rendered is dependent on the well object's *visible* property. When the component has been loaded into the DOM, the well data is fetched from the well service. The X3DOM code for each well is then generated and appended to a string variable. When X3DOM code for all wells has been generated, the string containing the code is inserted into the *<transform>* element of the Angular component using JQuery. The X3DOM code generation function for the well shape is where the differences from the slice visualization are situated, and will be explained in more detail below.

Since the well object has an array of positions and property values representing a certain segment of the well, the visualization is composed of a set of cylinder shapes, where each cylinder represents a well segment. The property values are visualized by coloring the cylinder according to the value. This visualization can be seen in *Figure 19* below.

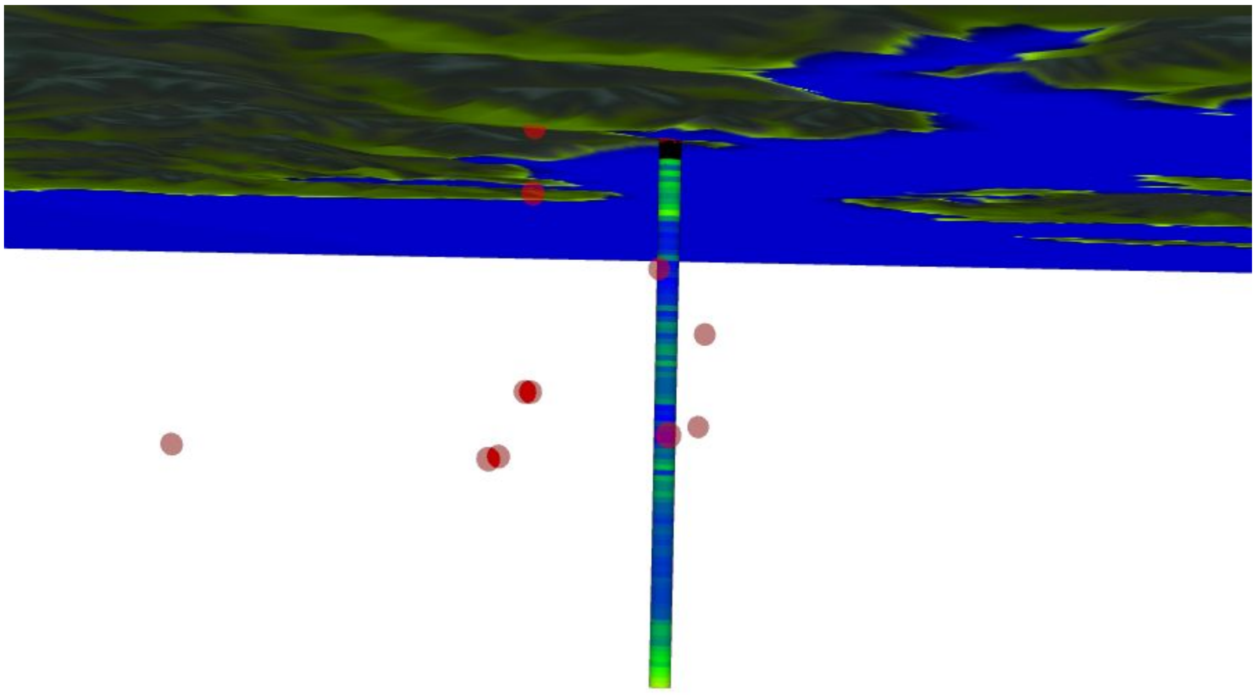


Figure 19: Visualization of the porosity of the Dh1 borehole. The color coding is from purple (low) to red (high). All slices in the scene has been toggled off, and only substitute geometry is displayed.

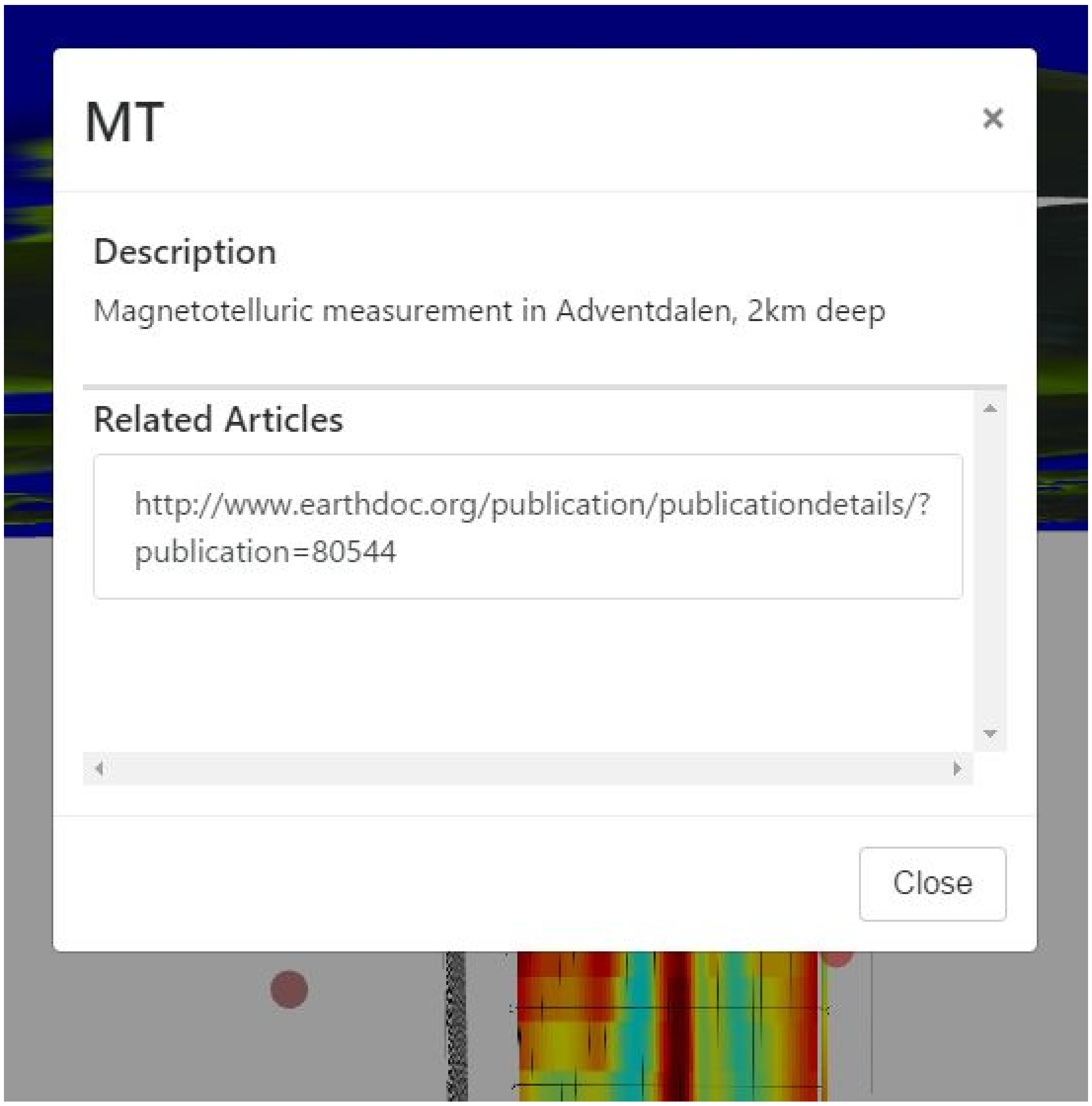
To generate the X3DOM code for the cylinder shapes, a height, position and color is required. An object with arrays for each of these properties, for each well segment, is created in a separate function. This function loops over the positions array of the well, and uses the depth value of the current and next position to calculate the height of the well segment. The depth of the position for the cylinder shape is translated by half the height of the segment, so that the positions represent the top of the cylinder.

The color coding is calculated by using the visualization options found on the well object. These options include the minimum and maximum value of the current property to be visualized. Based on this, the value is placed as a percentage between the minimum and maximum values. This percentage is used to sample the color of a 100 pixels wide, 1 pixel high HTML5 canvas element that has a gradient drawn upon it. It is possible with the JavaScript API for the HTML5 canvas to create and draw on a canvas element that is only residing in memory. The canvas element is created this way when the Well component is initialized. The canvas API is used to draw a gradient on the canvas. The API also has a function for reading the RGBA values of a canvas element at a specific coordinate. This function is used to sample the color coding of property values, using the calculated value percentage as the x-position on the canvas.

When the necessary information has been created, the code for a cylinder shape is generated for each well segment of the well. The code for the cylinders is surrounded with a *<transform>* element on which an *id* attribute is added, so that the individual wells can be accessed by DOM functions. After the X3DOM code for the wells has been inserted into the Angular component's view, event listeners for each individual cylinder shape for each well is added. If a segment of a well is left-clicked, its *visible* property is inverted. The well shape is then removed from the DOM, and replaced with the substitute shape. If a segment of well is right-clicked, the well information modal window opens up, and the color of the clicked segment is passed to the modal service, letting the user view the value of the visualized property at the point of the click.

4.4 Information modal window

The information modal is the window that appears when a shape in the application is right-clicked. It displays information about the data that is visualized, such as a textual description, and URLs to previously published related articles. For the well shape, this information window also contains the controls for the visualization of the well itself. *Figure 20* below shows the information window for a slice.



MT

Description

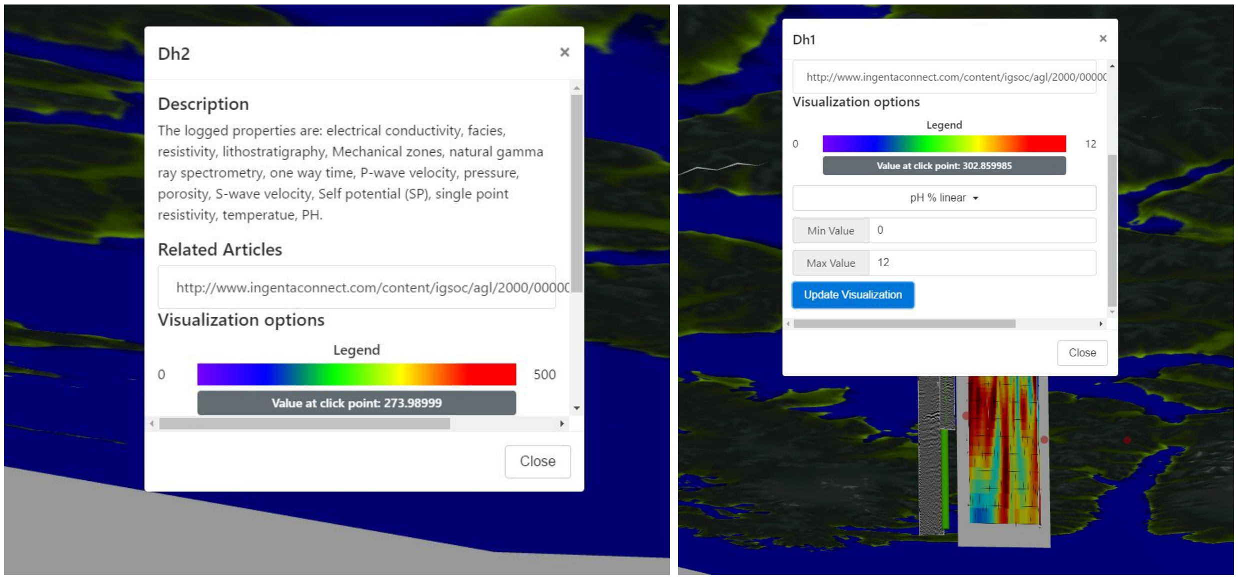
Magnetotelluric measurement in Adventdalen, 2km deep

Related Articles

http://www.earthdoc.org/publication/publicationdetails/?publication=80544

Figure 20: The slice information modal window. It shows the description of the slice data, along with a list of links to related publications.

In *Figure 21* below, the visualization options for a well can be seen. There is a color legend, showing how the color changes from minimum value to maximum value. Below that, the value of the visualization at the point of the right-click that opened the modal window can be seen. This has been recalculated from color value to decimal value, and lets the user view the value at a particular point in the visualization of the selected property for the well. The select menu below the legend and the click value, lets the user select which property of the well to visualize. It is also necessary to specify minimum and maximum values for the property. The user can then update the visualization, and when the window is closed, the changes to the visualization will take effect.


Dh2

Description

The logged properties are: electrical conductivity, facies, resistivity, lithostratigraphy, Mechanical zones, natural gamma ray spectrometry, one way time, P-wave velocity, pressure, porosity, S-wave velocity, Self potential (SP), single point resistivity, temperatue, PH.

Related Articles

http://www.ingentaconnect.com/content/igsoc/agl/2000/0000C

Visualization options

Legend

0 500

Value at click point: 273.98999

Dh1

http://www.ingentaconnect.com/content/igsoc/agl/2000/0000C

Visualization options

Legend

0 12

Value at click point: 302.859985

pH % linear

Min Value	0
Max Value	12

Figure 21: The well information modal window, with the top displayed on the left, and the bottom containing the well visualization settings to the right.

4.5 Server and Database

A database for storing the information to be visualized and a server to access the database is needed. For this, a Node.js server application, using MongoDB as a database was used. These technologies were selected because they are part of a common technology stack accompanied with Angular called MEAN (MongoDB, Express.js, Angular and Node.js), and work well together (as stated in [30, 31, 32, 33]).

Node.js [34] is a runtime for JavaScript, that lets the language be used outside the web browser. Node.js has a large amount of open source packages available in its package system called npm (Node Package Manager) [34]. One of these packages is called Express [35]. It is a minimalistic web application framework for Node.js, that provides HTTP utilities and functions making it easy to create REST API's [35].

"REST defines a set of architectural principles by which you can design Web services that focus on a system's resources" [28]. RESTful web service resources are addressed using URIs, and HTTP methods are used explicitly to manipulate the resource. The HTTP GET method is used to retrieve a resource, the POST method is used to create a new resource, the PUT method to update a resource, and the DELETE method to delete a resource [28].

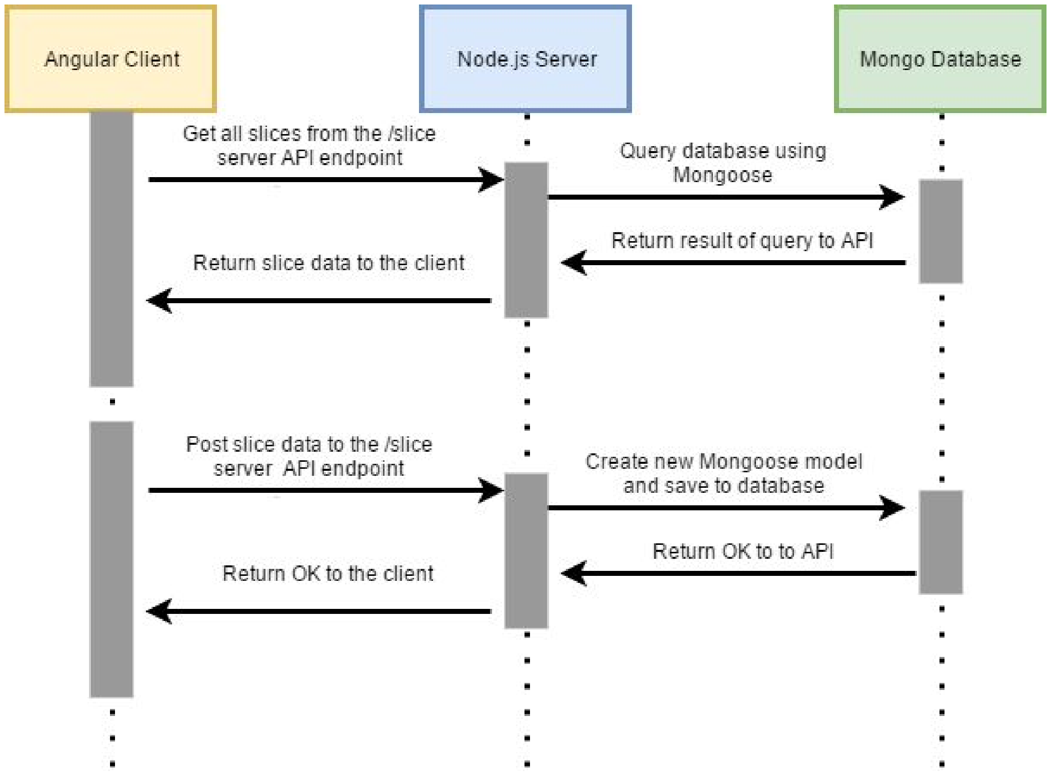
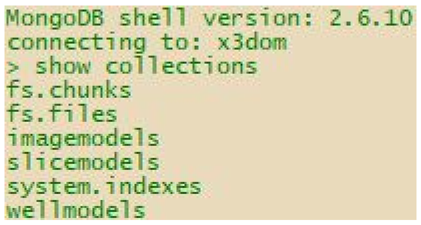

Angular Client
Node.js Server
Mongo Database
Get all slices from the /slice server API endpoint
Query database using Mongoose
Return result of query to API
Return slice data to the client
Post slice data to the /slice server API endpoint
Create new Mongoose model and save to database
Return OK to to API
Return OK to the client

Figure 22: Examples of the interaction between the client, server and database.

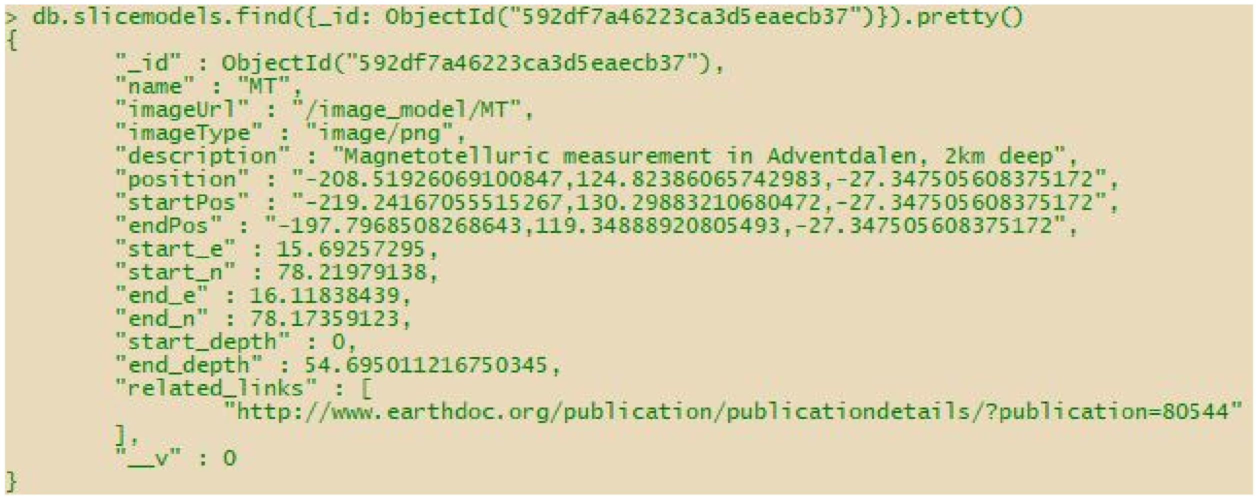
In *Figure 22* above, some interactions between the client, server and database can be seen. The client application uses the REST API of the server to fetch and upload data to the database. The server application contacts the database via the Mongoose package, and returns or uploads the data, respectively.

Node.js and Express are used together to create the server application. The only purpose of this application is to read and write information to the database, as requested by the client application. Another npm package, Mongoose [36], is used for communicating with the MongoDB database from the server. Mongoose provides a simple way to model data to be stored in a MongoDB database. It also has functionality for easy interaction with, and query of, the database [36]. MongoDB is a NoSQL database, meaning that instead of traditional SQL table row structures, MongoDB stores data as *documents*. A document in MongoDB is a data structure composed of field and value pairs. Documents are stored in collections, which can be seen as the MongoDB equivalent of SQL tables. An advantage of documents when used with JavaScript is that they are directly mappable to JavaScript objects. MongoDB uses a query language that resembles JavaScript to fetch documents [52]. An example of querying the database via the mongo shell application can be seen in *Figure 23 & 24*.



```
MongoDB shell version: 2.6.10
connecting to: x3dom
> show collections
fs.chunks
fs.files
imagemodels
slicemodels
system.indexes
wellmodels
```

Figure 23: The mongo shell application used to display all collections in the database.



```
> db.slicemodels.find({_id: ObjectId("592df7a46223ca3d5eaecb37")}).pretty()
{
        "_id" : ObjectId("592df7a46223ca3d5eaecb37"),
        "name" : "MT",
        "imageUrl" : "/image_model/MT",
        "imageType" : "image/png",
        "description" : "Magnetotelluric measurement in Adventdalen, 2km deep",
        "position" : "-208.51926069100847,124.82386065742983,-27.347505608375172",
        "startPos" : "-219.24167055515267,130.29883210680472,-27.347505608375172",
        "endPos" : "-197.7968508268643,119.34888920805493,-27.347505608375172",
        "start_e" : 15.69257295,
        "start_n" : 78.21979138,
        "end_e" : 16.11838439,
        "end_n" : 78.17359123,
        "start_depth" : 0,
        "end_depth" : 54.695011216750345,
        "related_links" : [
                "http://www.earthdoc.org/publication/publicationdetails/?publication=80544"
        ],
        "__v" : 0
}
```

Figure 24: The mongo shell application used to show the database representation of a slice.

5 Conclusion

We have created an application for visualizing subsurface geological data in 3D in web browsers, using the X3DOM framework. The application lets users upload seismic slice images and well logs, along with accompanying meta information, to the application's database. These are visualized in a scene along with a topography of Svalbard. The user can navigate around in a 3D landscape and go underground to view the visualized seismic slices and well visualizations. The seismic slices are visualized on vertical planes in the scene, and can be toggled off by clicking the plane. A red sphere is then displayed at the position of the slice, indicating that there is data at this point, but currently it is hidden. The slice can also be right-clicked, which will bring up a modal window with information about the slice, along with a list of links to related publications. In addition to direct interactions, the slices can be controlled by using the sidebar menu. This menu contains a list of the slices, along with checkboxes indicating whether each slice is visible or hidden. The menu can also be used to navigate directly to a specific slice by pressing the view button, or to open the information window by pressing the info button.

The wells are visualized as cylindrical tubes. The values of a user-selected property from the well log are visualized as colors on the tubes. Users can interact with the visualization by clicking on the well. This will cause the well visualization to be toggled off, showing a sphere in its place. Right-clicks on the well opens an information modal window. This window displays information about the well and links to related publications. The information window lets the user control the visualization of each individual well, and it also displays the property value of the well at the position of the right-click, letting the user get an exact value from the visualization.

The application was created using the X3DOM graphics framework, along with the Angular web framework. It was attempted to create reusable Angular components that wrapped logical groupings of X3DOM code, and controlled interactions with the X3DOM shapes declared in this code. This did not succeed as sometimes a "cannot read property of undefined" error was thrown. The reason for this error was hard to isolate, but appears to be caused by Angular's destruction of components. To solve this problem, Angular components were used with a single X3DOM *<transform>* element, declared with an *id* attribute. Inside the component, the element is accessed programmatically using DOM functionality. The X3DOM code for the visualization is generated from data that is loaded from the server via an Angular service. The code is then inserted into the *<transform>* element when the component has loaded. This approach enables the creation of "modules" containing groupings of X3DOM code and interaction logic for a subset of the scene, that can potentially be reused in other applications. It also allows the

X3DOM part of the application to naturally interact with the application state of Angular, using services that can communicate seamlessly with the rest of the application.

In conclusion, the application created in this thesis shows that it is possible to interactively visualize 3D subsurface geological data using X3DOM. Users can access the web application from a variety of platforms and devices supported by X3DOM, and quickly navigate to interesting data points using the application's viewpoint functionality. This, along with the rendered topography surrounding the data, allows users to get a quick overview of available data in the area, thus fulfilling the main requirement of the application.

5.1 Discussion

In this thesis visualizations were built by combining existing X3DOM nodes. However X3DOM lets developers define their own X3DOM nodes with custom appearance and functionality. Therefore the well and slice objects could have been represented as two custom and reusable X3DOM nodes. Custom nodes are created using X3DOM's JavaScript API, which can make use of WebGL and GLSL shaders. However, one of the reasons for choosing X3DOM as a graphics framework in this thesis was to achieve a high level of abstraction through declarative programming and avoid low level WebGL programming. There exists several libraries that encapsulate WebGL such as Three.js which makes programming simpler, however these libraries still require the user to understand Javascript and large imperative APIs instead of declaring and attaching high-level X3DOM nodes with well defined and small interfaces. Therefore the approach of creating custom X3DOM nodes was not investigated. We believe that X3DOM is well suited for situations where the application can be built using already existing nodes. If this is not possible we believe that using an imperative framework with direct access to the underlying technologies (such as WebGL or Three.js) requires less work than creating custom X3DOM nodes. This is because the latter requires first implementing the visualization with an imperative framework and then encapsulating the solutions into an X3DOM node. On the other hand, if the custom nodes that need to be created have a high level of reusability, and the project consists of several programmers, it might still be better to invest the extra time into creating these nodes. This would require a small selection of programmers to perform, possibly time consuming, low-level coding and create a high-level library of custom X3DOM nodes. Then the rest of the team would be users of the library and would not need knowledge of low-level APIs such as WebGL. X3DOM is an excellent choice of framework, that requires very little knowledge outside of standard DOM manipulation.

6 Future Work

The development of the application using X3DOM directly inside Angular components seemed promising in the beginning, but was found to produce errors. The reasons for this are not clear, and should be investigated more thoroughly. If Angular and X3DOM could be used together seamlessly, custom components with X3DOM functionality could be created without having to resort to X3DOM's JavaScript API.

A useful extension of the application's functionality would be to encode camera transformation and visualization settings in a URL string. This would allow users to copy their URL and share it with colleagues who can then paste it into their browser and see exactly the same, thus fostering collaboration.

Currently, the application uses a static set of textures for the BVHRefiner node of X3DOM which creates the terrain. Going forward, it would be beneficial to add support that would let users dynamically upload GeoTIFF files for new areas. The file could then be used by the server to generate the necessary texture files for an instance of the BVHRefiner node, which could then render the topology from the GeoTIFF file. Also, being able to toggle the visibility of terrains would allow users to upload their own versions of terrains with custom texture projections such as rock lithology. However, such a feature would possibly be time-consuming and complex to implement. It was therefore deemed out of scope for the thesis.

7 Appendices

Appendix A

Excerpt of React's source code which shows how the library bypasses the attribute whitelist when the *is* attribute is added to an element [3].

```
function isCustomComponent(tagName, props) {
  return tagName.indexOf('-') >= 0 || props.is != null;
}

.
.
.

if (this._tag != null && isCustomComponent(this._tag, props)) {
  if (!RESERVED_PROPS.hasOwnProperty(propKey)) {
        markup = DOMPropertyOperations.createMarkupForCustomAttribute(propKey, propValue);
  }
}
```

8 Bibliography

[1] J. Behr, P. Eschler, Y. Jung and M. Zöllner, "X3DOM: a DOM-based HTML5/X3D integration model", *Web3D '09 Proceedings of the 14th International Conference on 3D Web Technology*, 2009.

[2] J. Dirksen, *Learning Three.js: The JavaScript 3D Library for WebGL,*, 1st ed. Birmingham: Packt Publishing, 2015, pp. 7-35, 37-64.

[3] "ReactDOMComponent.js", *React source code at GitHub*. [Online]. Available: https://github.com/facebook/react/blob/6beb87eb73cb7a5e390c47614b5f796afdfa3f65/src/renderers/dom/stack/client/ReactDOMComponent.js. [Accessed: 31- May- 2017].

[4] T. Parisi, *Programming 3d applications with html5 and webgl*, 1st ed. O'Reilly Media, Inc, 2014, pp. 17-28.

[5] Tavakoli, Langeland and Patel, "VIRCOLA - Review of Data and Visualization Platform (Internal report)", CMR, Bergen, 2013.

[6] Langeland and Patel, "VIRCOLA – Virtual CO2 Laboratory (Information poster)", CMR, Bergen, 2014.

[7] Krämer and Gutbell, "A case study on 3D geospatial applications in the Web using state-of-the-art WebGL frameworks", *Web3D '15 Proceedings of the 20th International Conference on 3D Web Technology*, pp. 189-197, 2015.

[8] J. Ziolkowska and R. Reyes, "Geological and hydrological visualization models for Digital Earth representation", *Computers & Geosciences*, vol. 94, pp. 31-39, 2016.

[9] Pano, Graziotin and Abrahamsson, "Rationale leading to the adoption of a JavaScript framework", https://arxiv.org/pdf/1605.04303.pdf, 2016.

[10] "Angular Documentation, Architecture", *Angular.io*, 2017. [Online]. Available: https://angular.io/docs/ts/latest/guide/architecture.html, [Accessed: 31- May- 2017].

[11] "React - A JavaScript library for building user interfaces", *Facebook.github.io*, 2017. [Online]. Available: https://facebook.github.io/react. [Accessed: 31- May- 2017].

[12] "Array.prototype.map()", *Mozilla JavaScript Reference*, 2017. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map. [Accessed: 31- May- 2017].

[13] "Introducing JSX - React", *Facebook.github.io*, 2017. [Online]. Available: https://facebook.github.io/react/docs/introducing-jsx.html. [Accessed: 31- May- 2017].

[14] "W3C Document Object Model", *W3.org*, 2017. [Online]. Available: https://www.w3.org/DOM/#what. [Accessed: 31- May- 2017].

[15] "Getting Started with X3D | Web3D Consortium", *Web3d.org*, 2017. [Online]. Available: http://www.web3d.org/getting-started-x3d. [Accessed: 31- May- 2017].

[16] "X3DOM at a Glance", *X3DOM*, 2017. [Online]. Available: https://www.x3dom.org/. [Accessed: 31- May- 2017].

[17] "BVHRefiner", *X3DOM Documentation*, 2017. [Online]. Available: https://doc.x3dom.org/tutorials/largeModels/BVHRefiner/index.html. [Accessed: 31- May- 2017].

[18] "Terrengmodell Svalbard (S0 Terrengmodell)", *Data.npolar.no*, 2017. [Online]. Available: https://data.npolar.no/dataset/dce53a47-c726-4845-85c3-a65b46fe2fea. [Accessed: 31- May- 2017].

[19] Zhao Y. “Towards Open Source Remote Sensing Software – A Survey”. In: Bian F., Xie Y., Cui X., Zeng Y. (eds) Geo-Informatics in Resource Management and Sustainable Ecosystem. Communications in Computer and Information Science, vol 398. Springer, Berlin, Heidelberg, 2013.

[20] "Dependency Injection", *Angular Documentation*, 2017. [Online]. Available: https://angular.io/docs/ts/latest/guide/architecture.html#!#dependency-injection. [Accessed: 31- May- 2017].

[21] "Services", *Angular Documentation*, 2017. [Online]. Available: https://angular.io/docs/ts/latest/guide/architecture.html#!#services. [Accessed: 31- May- 2017].

[22] "HTTP Client", *Angular Documentation*, 2017. [Online]. Available: https://angular.io/docs/ts/latest/guide/server-communication.html. [Accessed: 31- May- 2017].

[23] "RxJS", *RxJS Documentation*, 2017. [Online]. Available: http://reactivex.io/rxjs/. [Accessed: 31- May- 2017].

[24] "RxJS Library", *Angular Documentation*, 2017. [Online]. Available: https://angular.io/docs/ts/latest/guide/server-communication.html#!#rxjs-library. [Accessed: 31- May- 2017].

[25] "Observable", *ReactiveX Documentation*, 2017. [Online]. Available: http://reactivex.io/documentation/observable.html. [Accessed: 31- May- 2017].

[26] "Subject", *ReactiveX Documentation*, 2017. [Online]. Available: http://reactivex.io/documentation/subject.html. [Accessed: 31- May- 2017].

[27] "Papa Parse", *Papa Parse*, 2017. [Online]. Available: http://papaparse.com/. [Accessed: 31- May- 2017].

[28] "jQuery", *jQuery*, 2017. [Online]. Available: http://jquery.com/. [Accessed: 31- May- 2017].

[29] "RESTful Web services: The basics", *IBM developerWorks*, 2017. [Online]. Available: https://www.ibm.com/developerworks/library/ws-restful/. [Accessed: 31- May- 2017].

[30] "MEAN App with Angular 2 and the Angular CLI", *Scotch.io*, 2017. [Online]. Available: https://scotch.io/tutorials/mean-app-with-angular-2-and-the-angular-cli. [Accessed: 31- May- 2017].

[31] "Developing a MEAN app with Angular 2.0", *The Jackal of Javascript*, 2017. [Online]. Available: http://thejackalofjavascript.com/developing-a-mean-app-with-angular-2-0/. [Accessed: 31- May- 2017].

[32] "Angular 2 (or 4) & NodeJS - The Practical MEAN Stack Guide", *Udemy*, 2017. [Online]. Available: https://www.udemy.com/angular-2-and-nodejs-the-practical-guide/. [Accessed: 31- May- 2017].

[33] "Creating a MEAN Stack with Angular 2 and TypeScript", *Medium*, 2017. [Online]. Available: https://medium.com/@tsmith18256/creating-a-mean-stack-with-angular-2-and-typescript-3dd23b3e717f. [Accessed: 31- May- 2017].

[34] "About | Node.js", *Nodejs.org*, 2017. [Online]. Available: https://nodejs.org/en/about/. [Accessed: 31- May- 2017].

[35] "Express - Node.js web application framework", *Expressjs.com*, 2017. [Online]. Available: https://expressjs.com/. [Accessed: 31- May- 2017].

[36] "Mongoose ODM v4.10.4", *Mongoosejs.com*, 2017. [Online]. Available: http://mongoosejs.com/. [Accessed: 31- May- 2017].

[37] "Petrel E&P Software Platform", *Software.slb.com*, 2017. [Online]. Available: http://www.software.slb.com/products/petrel. [Accessed: 31- May- 2017].

[38] "SKUA® Software Suite by Paradigm®", *Pdgm.com*, 2017. [Online]. Available: http://www.pdgm.com/products/skua-gocad/. [Accessed: 31- May- 2017].

[39] "Browser support - x3dom.org", *X3DOM*, 2017. [Online]. Available: https://www.x3dom.org/contact/. [Accessed: 31- May- 2017].

[40] "Angular", *Angular.io*, 2017. [Online]. Available: https://angular.io/. [Accessed: 31- May- 2017].

[41] Nimtz, M., Klatt, M., Wiese, B., Kühn, M., Krautz, H.-J., 2010. Modelling of the CO2 process- and transport chain in CCS system-examination of transport and storage processes. – Chemie der Erde –Geochemistry, 70, Suppl. 3, pp. 185-192.

[42] "Hello, X3DOM", *X3DOM Documentation*, 2017. [Online]. Available: https://doc.x3dom.org/tutorials/basics/hello/index.html. [Accessed: 31- May- 2017].

[43] "Picking Objects", *X3DOM Documentation*, 2017. [Online]. Available: https://doc.x3dom.org/tutorials/animationInteraction/picking/index.html. [Accessed: 31- May- 2017].

[44] "JSX In Depth", *React Documentation*, 2017. [Online]. Available: https://facebook.github.io/react/docs/jsx-in-depth.html. [Accessed: 31- May- 2017].

[45] "Template Syntax", *Angular Documentation*, 2017. [Online]. Available: https://angular.io/docs/ts/latest/guide/template-syntax.html#!#other-bindings. [Accessed: 31- May- 2017].

[46] "ElementRef", *Angular Documentation*, 2017. [Online]. Available: https://angular.io/docs/js/latest/api/core/index/ElementRef-class.html. [Accessed: 31- May- 2017].

[47] "Tags", *Stackoverflow.com*, 2017. [Online]. Available: https://stackoverflow.com/tags?page=3&tab=popular. [Accessed: 31- May- 2017].

[48] "Metadata", *Angular Documentation*, 2017. [Online]. Available: https://angular.io/docs/ts/latest/guide/architecture.html#!#components. [Accessed: 31- May- 2017].

[49] "Optimizing Performance. Avoid Reconciliation.", *React Documentation*, 2017. [Online]. Available: https://facebook.github.io/react/docs/optimizing-performance.html. [Accessed: 31- May- 2017].

[50] "Unknown Prop Warning", *React Documentation*, 2017. [Online]. Available: https://facebook.github.io/react/warnings/unknown-prop.html. [Accessed: 31- May- 2017].

[51] "True custom attributes (e.g. Microdata) in React" (answer by Christian Steinmann 22.11.2015), *Stackoverflow.com*, 2017. [Online]. Available: https://stackoverflow.com/questions/21648347/true-custom-attributes-e-g-microdata-in-react/33860892#33860892. [Accessed: 31- May- 2017].

[52] "Introduction to MongoDB", *MongoDB Manual*, 2017. [Online]. Available: https://docs.mongodb.com/manual/introduction/. [Accessed: 31- May- 2017].

9 Glossary

API - Application Programming Interface.
Callback (function) - A function that is passed as a parameter to another function, which executes the passed function instead of returning a value.
Canvas - The HTML5 canvas element.
Component - A structure that contains and controls a piece of UI. React and Angular uses components to build applications.
CSV - Comma Separated Value filetype.
DI - Dependency Injection.
Attribute Directive (Angular) - A structure that can be added to elements, changing their appearance or behavior.
DOM - HTML Document Object Model, which allows JavaScript to access and manipulate the elements of an HTML document.
Geological slice - 2D section of geology, represented by an image.
Geometry - Collective name of triangles a graphical shape consists of.
Geospatial - Data with positional information relative to the earth's surface.
HTML element/node/tag - Tag is used to describe the textual representation of the element in the HTML document, while element/node is used interchangeably for referring to an element in a rendered HTML page/DOM tree.
JSON - JavaScript Object Notation.
Lifecycle event - Events related to the lifecycle of components, such as when they are instantiated, added to the HTML page, and destructed.
LOD - Level of detail.
Observable - An RxJS Observable, which is an object that can be provided with a callback function, which will then be notified of events on the Observable.
RMS File - Textual file containing well log data.
Scene - X3DOM's scene node, providing scenegraph functionality.
Service (Angular) - A regular class which can be injected into components to provide functionality that is not related to the display of data.
Shader - Small programs that runs on the GPU, using data in the GPU's video memory to process, shade and render geometry.
Template (Angular) - a template describes the HTML elements in a component's view.
TypeScript - A superset of JavaScript, providing features from future JavaScript standards not yet implemented in most browsers, and static type checking.
UI - User Interface
UTM - Universal Transverse Mercator coordinate system.

View - The rendered UI of a component (Angular/React) or a specific part of the rendered HTML page.
X3DOM element/node/tag Tag is used to describe the textual representation of the element in the X3DOM code, while element/node is used interchangeably for referring to an element in a rendered X3DOM scene.